TEXAS INSTRUMENTS

FPGA

Applications Handbook

1993

Application Specific Products

Printed on
ENVIRONMENTALLY FRIENDLY OFFSET
Cover on
ENVIROCOTE 250GSM BOARD

FPGA Applications Handbook

IMPORTANT NOTICE

Texas Instruments Incorporated (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to current specifications in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Please be aware that TI products are not intended for use in life-support appliances, devices, or systems. Use of TI product in such applications requires the written approval of the appropriate TI officer. Certain applications using semiconductor devices may involve potential risks of personal injury, property damage, or loss of life. In order to minimize these risks, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards. Inclusion of TI products in such applications is understood to be fully at the risk of the customer using TI devices or systems.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

WARNING

This equipment is intended for use in a laboratory test environment only. It generates, uses, and can radiate radio frequency energy and has not been tested for compliance with the limits of computing devices pursuant to subpart J of part 15 of FCC rules, which are designed to provide reasonable protection against radio frequency interference. Operation of this equipment in other environments may cause interference with radio communications, in which case the user at his own expense will be required to take whatever measures may be required to correct this interference.

Copyright © 1993, Texas Instruments Incorporated

Preface

Read This First

This handbook serves as a reference on FPGA technology. Study of this book will provide you with an in-depth knowledge of FPGA design.

Document Contents

This document contains the following chapters:

Chapter 1 Introduction

Chapter 1 explores the development of the field programmable gate array and introduces the FPGA architecture, applications, and design benefits.

Chapter 2 FPGA Design Flow

Chapter 2 discusses the FPGA design flow for various software utilities and supported hardware platforms.

Chapter 3 TI Action Logic System

Chapter 3 discusses the development system for configuring, programming, and debugging TI FPGAs. The system consists of programming hardware and software which operates on both personal computers and workstations.

Chapter 4 FPGA Logic Synthesis and Modeling

Chapter 4 describes logic synthesis and modeling as an alternative to schematic entry for designing an electronic circuit or system.

Chapter 5 PLD to FPGA Migration

Chapter 5 addresses methods and issues regarding migration to and from FPGAs. For PAL designers, FPGAs represent the next step in logic consolidation for reduction of board area, power consumption, and cost. Also, migration to hardwired gate arrays is an option for high volume designs or consolidation of several FPGAs.

Chapter 6 FPGA to ASIC Migration

Chapter 6 describes the automated methodology available for migrating field programmable logic (FPL) to gate array or standard cell ASIC technology. Options for migrating FPGA to ASIC are discussed, along with migration design rules for testability.

Chapter 7 FPGA Programming and Test

This chapter describes programming and test issues. FPGA test and programming can take several different forms depending upon the design volumes and stage of the design. Factory programmed and tested devices are an option for large volume designs and the related technical and business issues are described. In the early stages of design, other methods are more desirable and these are discussed here also.

Chapter 8 Application Examples

This chapter presents specific TI FPGA design applications including JTAG implementation, a microcontroller design, pipelined design, and serial communications interface.

Style and Symbol Conventions

This document uses the following conventions.

- Program listings, program examples, interactive displays, system messages, filenames, and directory names appear in a `font like this.`
- Instructions, commands, directives, and information typed in response to system prompts appear in **boldface** and parameters appear in ***bold-faced italics***. Portions of a syntax that are in **boldface** should be entered as shown; portions of a syntax that are in ***bold-faced italics*** indicate variables and describe the type of information that should be entered. Here is an example of a directive syntax:

  makeadl ***design_name***

  function_name ***design_name***

- Document titles, chapters and sections in this user's guide or in other referenced documents are shown in italics. Other words of emphasis in this guide are also shown in italics.
- Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets; you don't enter the brackets themselves. Here's an example of an instruction that has an optional parameter:

  LALK ***16-bit constant [, shift]***

  The **LALK** instruction has two parameters. The first parameter, *16-bit constant,* is required. The second parameter, *shift,* is optional. As this

syntax shows, if you use the optional second parameter, you must precede it with a comma.

- Braces ({ and }) indicate a list. The symbol | (read as or) separates items within the list. Here's an example of a list:

  { * | *+ | *- }

  This provides three choices: *, *+, or *-.

  Unless the list is enclosed in square brackets, you must choose one item from the list.

- Some directives can have a varying number of parameters. For example, the **.byte** directive can have up to 100 parameters. The syntax for this directive is:

  .byte *value1 [, ... , valuen]*

  This syntax shows that **.byte** must have at least one value parameter, but you have the option of supplying additional value parameters, separated by commas

Information about Cautions and Warnings

This book may contain cautions and warnings.

> **This is a caution.**
>
> A **caution** describes a situation that could potentially damage your software or equipment.

> **This is a warning.**
>
> A **warning** describes a situation that could potentially cause harm to you.

The information in a caution or a warning is provided for your protection. Please read each caution and warning carefully.

Related Documentation

TI-ALS 2.2 User's Guide

FPGA 2.2 Data Manual

TPC10 Series Family Data Sheet

TPC12 Series Family Data Sheet

Trademarks

ABEL is a trademark of Data I/O Corporation.
Action Logic, Actionprobe, Activator, ACT, ALES, APS, and *PLICE* are trademarks of Actel Corporation.
AEGIS is a trademark of AEGIS Development, Inc.
Apollo, HP/Apollo, and *Domain* are trademarks of Hewlett Packard Company.
Cadence, Verilog, and *VerilogXL* are trademarks of Cadence Design Systems, Inc.
CUPL is a trademark of Personal CAD Systems, Inc.
LOG/IC is a trademark of ISdata Incorporated.
Logitech is a trademark of Logitech, Incorporated.
Mentor Graphics, NetEd, SymEd, QuickSIM, and *ESIread* are trademarks of Mentor Graphics, Inc.
MS-DOS is a trademark of Microsoft Corporation.
Norton Commander is a trademark of Symantec Corporation.
OrCAD is a trademark of OrCAD Systems Corporation.
PAL and *PALASM* are trademarks of Advanced Micro Devices, Inc.
P-CAD is a trademark of Personal CAD Systems, Inc.
PGADesigner, PLDesigner, and *PLDesigner-XL* are trademarks of Minc Incorporated.
proLogic is a trademark of proLogic Systems, Inc..
Racal-Redac is a trademark of Racal Electronics Company.
RapidSIM is a trademark of Valid Logic Systems Incorporated.
Sophia is a trademark of Sophia Computer Systems, Inc.
Sun, Sun Workstation, and *SPARCstation* are trademarks of Sun Microsystems, Inc.
Synopsys is a trademark of Synopsys, Inc.
Teradyne is a trademark of Teradyne, Inc.
Unisite is a trademark of Data I/O Corporation.
UNIX is a trademark of AT&T Bell Laboratories, Inc.
Viewdraw, Viewfile, Viewgen, Viewlogic, Viewsim, Viewwave, and *Workview* are trademarks of Viewlogic Systems, Incorporated.

Contents

1 Introduction 1-1

1.1 Field Programmable Gate Arrays—Revolutionizing Logic Design 1-2
1.1.1 General Purpose Logic 1-2
1.1.2 Programmable Logic Devices 1-3
1.1.3 Field Programmable Gate Arrays 1-8
1.1.4 Application Specific Integrated Circuits 1-12

1.2 FPGA Device Architecture 1-15
1.2.1 Introduction 1-15
1.2.2 Architectural Overview 1-16
1.2.3 TPC10 Series Specific Features 1-19
1.2.4 TPC12 Series Specific Features 1-20
1.2.5 Generating Logic Functions from Multiplexers 1-22

2 FPGA Design Flow 2-1

2.1 Viewlogic on PC 2-2
2.1.1 Introduction 2-2
2.1.2 Viewlogic Design Flow 2-2
2.1.3 Creating a Schematic 2-4
2.1.4 Simulating a Schematic 2-8
2.1.5 Generating an ADL Netlist 2-14
2.1.6 Postlayout Simulation 2-14

2.2 OrCAD on PC 2-15
2.2.1 Introduction 2-15
2.2.2 PC Requirements 2-15
2.2.3 `Autoexec.bat` and `config.sys` 2-15
2.2.4 OrCAD Design Flow 2-16
2.2.5 Invoking OrCAD 2-18
2.2.6 Establishing a New Design 2-18
2.2.7 Design Entry and Schematic Capture 2-18
2.2.8 Logic Simulation 2-22
2.2.9 Netlist Translation 2-23

2.3 Valid on Sun 2-31
2.3.1 Introduction 2-31
2.3.2 Directory Structure and Symbolic Links 2-33
2.3.3 TI-ALS/Valid Library Files 2-33
2.3.4 ValidGED Schematic Editor 2-33
2.3.5 TI-ALS Conventions for Schematic Capture Using ValidGED 2-34
2.3.6 Adding Title and Abbreviation Properties 2-34

2.3.7 Modifying TI-ALS Soft Macros 2-35
2.3.8 Top-Level Symbols 2-35
2.3.9 Unit-Delay Functional Simulation 2-35
2.3.10 ADL Netlist Conversion 2-35
2.3.11 Postroute Simulation with Back Annotated Delays 2-35
2.4 Viewlogic on Sun 2-37
2.4.1 Introduction 2-37
2.4.2 Symbolic Links 2-39
2.4.3 Directory Structure 2-39
2.4.4 TI-ALS/Viewlogic Library Files 2-40
2.4.5 TI-ALS/Viewlogic Simulation Models 2-40
2.4.6 Specifying Libraries in the `viewdraw.ini` File (Workview 4.1) 2-40
2.4.7 UNIX Environment Requirements 2-40
2.4.8 Adding Power and Ground 2-41
2.4.9 Modifying TI-ALS Soft Macros 2-41
2.4.10 Adding Pins to the Design 2-41
2.4.11 Top-Level Symbol 2-42
2.4.12 Hierarchy—Sheets Versus Symbols 2-42
2.4.13 Functional Simulation 2-42
2.4.14 Creating the TI-ALS Netlist from the Schematic 2-42
2.4.15 Back Annotating Time Delays 2-43
2.4.16 Viewlogic Anomalies 2-43
2.5 Mentor on Apollo 2-44
2.5.1 Introduction 2-44
2.5.2 Design Overview 2-44
2.5.3 Design Flow—Schematic Capture 2-48
2.5.4 Design Verification 2-50
2.5.5 TI Action Logic System 2-53
2.5.6 FPGA Programming 2-57

3 TI Action Logic System 3-1
3.1 TI Action Logic System Overview 3-2
3.2 Configuring Workview for FPGA Libraries 3-9
3.2.1 Viewdraw Initialization File 3-9
3.2.2 Default Design Directory 3-9
3.2.3 Component Selection 3-10
3.3 Customizing Viewlogic Menus for the TI-ALS 3-11
3.3.1 Workview Menu Structure 3-11
3.3.2 Customizing Workview Menus for the TI-ALS 3-11
3.4 Understanding the ADL Netlist 3-17
3.4.1 Introduction 3-17
3.4.2 Example Design and Netlist 3-17
3.5 Back Annotated Simulation for FPGAs 3-23
3.5.1 Introduction 3-23
3.5.2 Prelayout Delay Estimations 3-23

3.5.3 Unit Delay Simulation ... 3-24
3.5.4 Postrouting Simulation ... 3-24

3.6 Multiple FPGA Simulations ... 3-28
3.6.1 Introduction ... 3-28
3.6.2 Single Chip Design ... 3-28
3.6.3 Multichip Design ... 3-29

3.7 Using the TI-ALS Timer for Static Timing Analysis ... 3-32
3.7.1 Introduction ... 3-32
3.7.2 Entering the Timer and Saving Timer Results ... 3-33
3.7.3 Prelayout Versus Postlayout ... 3-33
3.7.4 How the Timer Works ... 3-33
3.7.5 Creating and Modifying Sets ... 3-35
3.7.6 Changing Sets ... 3-37
3.7.7 Maximum Operating Frequency ... 3-37
3.7.8 Input-to-Output Delay ... 3-38
3.7.9 Setup and Hold Time Requirements ... 3-39
3.7.10 External Setup Time Requirements ... 3-40
3.7.11 Clock Skew ... 3-40
3.7.12 Pins Selection ... 3-41
3.7.13 Other Timer Options and Utilities ... 3-41

3.8 Critical Path Analysis for FPGAs ... 3-43
3.8.1 Methodology ... 3-43
3.8.2 Example 1—Combinational Path ... 3-45
3.8.3 Example 2—Sequential Critical Path ... 3-50

3.9 Understanding How TI-ALS Places and Routes ... 3-53
3.9.1 Introduction ... 3-53
3.9.2 Logic Module Vertical Tracks ... 3-53
3.9.3 Example Circuit ... 3-54
3.9.4 Pin Assignment, Pin File ... 3-56
3.9.5 Location File, Placement File ... 3-58

3.10 Using TI-ALS Debug ... 3-65
3.10.1 Introduction ... 3-65
3.10.2 Functional Debug ... 3-65
3.10.3 TI-ALS Debug Menu ... 3-66
3.10.4 Counter TA161 ... 3-67
3.10.5 Creating a Command File ... 3-69

3.11 Actionprobe for Functional Debug ... 3-74
3.11.1 Introduction ... 3-74
3.11.2 Actionprobe Setup ... 3-74
3.11.3 Configuring the Actionprobe ... 3-75
3.11.4 Actionprobe Resources ... 3-76

3.12 Net Criticality Within Workview ... 3-79
3.12.1 Introduction ... 3-79
3.12.2 Levels of Criticality ... 3-79
3.12.3 Define Critical Net ... 3-80

4 FPGA Logic Synthesis and Modeling 4-1

4.1 Logic Synthesis for FPGAs 4-2
- 4.1.1 Introduction 4-2
- 4.1.2 Design Methodology Trend 4-2
- 4.1.3 FPGA Logic Synthesis Tools 4-3
- 4.1.4 FPGA Logic Synthesis 4-4
- 4.1.5 FPGA Logic Synthesis with Synopsys Using VHDL/Verilog 4-8
- 4.1.6 Summary 4-12

4.2 ALES 1 – An FPGA Logic Enhancer/Synthesizer Tool for PC-386 4-13
- 4.2.1 Introduction 4-13
- 4.2.2 Function of ALES 1 4-14
- 4.2.3 ALES 1 Design Flow 4-14
- 4.2.4 ALES 1 Command Reference 4-16
- 4.2.5 Application Example (5-Bit Synchronous Counter) 4-17
- 4.2.6 Synthesis of a PALASM 2 Design 4-17
- 4.2.7 Optimizing an ADL Netlist 4-23
- 4.2.8 Analyzing Results Obtained from ALES 1 4-23
- 4.2.9 Summary 4-23

4.3 PLD, CPLD, and FPGA Design Flow Using PLDesigner-XL 4-24
- 4.3.1 Design Flow 4-24
- 4.3.2 Design Entry 4-26
- 4.3.3 Schematic Entry 4-26
- 4.3.4 Device Selection 4-27

4.4 Synopsys to FPGA Migration 4-30
- 4.4.1 Introduction 4-30
- 4.4.2 Synopsys To FPGA Flow 4-31
- 4.4.3 A Brief Overview of Synopsys 4-32
- 4.4.4 Compilation Strategy for Synopsys to FPGA Flow 4-33
- 4.4.5 Starting Synopsys 4-33
- 4.4.6 Specific Compilation For Source Files 4-34
- 4.4.7 Other Useful Tips 4-41
- 4.4.8 EDIF to Mentor Conversion 4-42
- 4.4.9 Mentor to QuickSIM 4-43
- 4.4.10 Running TI_NETED 4-43
- 4.4.11 Running the TI-ALS Software 4-44
- 4.4.12 Run Conversion Program 4-44
- 4.4.13 Timer 4-47
- 4.4.14 Back Annotation 4-47
- 4.4.15 Producing an FPGA 4-48
- 4.4.16 Activator 4-49
- 4.4.17 Actionprobe 4-49
- 4.4.18 Summary 4-49

4.5 12-Hour Clock – A High-Level Design Flow Example 4-50
- 4.5.1 Introduction 4-50
- 4.5.2 Design Overview 4-50

4.5.3 High-Level Design Flow 4-55
4.5.4 Design Synthesis 4-57
4.5.5 TI-ALS 4-61
4.5.6 Simulation 4-62
4.5.7 Summary 4-72
4.6 ExpressV-HDL 4-73

5 PLD to FPGA Migration 5-1
5.1 TI Logic Integration Tool (LIT) 5-2
5.1.1 Hardware Requirements 5-3
5.1.2 Libraries 5-4
5.1.3 Example 5-4
5.2 ProLogic Compiler 5-8
5.2.1 ProLogic Flow 5-8
5.2.2 Viewlogic Flow 5-12
5.2.3 TI-ALS Flow 5-15
5.2.4 A Complex Circuit With More Pins Than the Supported PLDs 5-17
5.3 FPGA Logic Synthesis Using Exemplar 5-20
5.3.1 Introduction 5-20
5.3.2 Software Description 5-20
5.3.3 Controlling Optimization 5-21
5.3.4 Other Features 5-21
5.3.5 Migration to ASIC 5-22
5.3.6 Design Examples 5-23

6 FPGA to ASIC Migration 6-1
6.1 Migrating Programmable Logic to ASIC 6-2
6.1.1 Motivation 6-2
6.1.2 Basic Requirements of Migration Methodology 6-3
6.1.3 Apply Good Digital Design Principles 6-4
6.1.4 FPGA Migration to ASIC 6-8
6.2 FPGA to ASIC Migration Options 6-11
6.2.1 Hardwired FPGAs 6-11
6.2.2 Preprogrammed FPGAs 6-11
6.2.3 FPGA Migration to ASIC 6-11
6.2.4 Migration Example 6-16
6.2.5 Logic Synthesis Migration Flow 6-19
6.3 Migration Design Rules for Testability 6-22
6.3.1 Latches 6-24
6.3.2 Gated Clocks 6-25
6.3.3 Clock Signals Used for Data 6-25
6.3.4 Using Both Edges of a Clock 6-26
6.3.5 Uncontrollable Asynchronous Pins 6-27
6.3.6 Q OUT Used to Clock a Flip-Flop 6-28
6.4 Concurrent FPGA and ASIC Design Using HDL 6-29

7 FPGA Programming and Test 7-1

7.1 Programming FPGAs - Considerations and Options 7-2
7.1.1 Programming Time and Throughput 7-2
7.1.2 Surface-Mount Component Lead Quality 7-5
7.1.3 Handling Moisture-Sensitive Plastic-Surface-Mount Components 7-7
7.1.4 Manufacturing Efficiency 7-8

7.2 How TI Tests FPGAs 7-11
7.2.1 Serial-Test Circuitry 7-11
7.2.1 Test Flow 7-12

7.3 Programming and Verifying FPGA Antifuses 7-15
7.3.1 Activator Programming Procedure Overview 7-15
7.3.2 Antifuse Programming 7-16
7.3.3 Antifuse Verification During Programming 7-16
7.3.4 Antifuse Integrity Test 7 and Integrity Test 8 7-17
7.3.5 Browsing Your .AVI File 7-17

7.4 Speed-Enhanced FPGA Devices 7-19
7.4.1 Why Speed Binning 7-19
7.4.2 TPC1010A and TPC1020A Binning Circuits 7-19
7.4.3 Setup and Measurement Procedure for the TPC10 Series 7-20
7.4.4 TPC12 Series Speed Binning 7-20

7.5 Automatic Test Pattern Generation (ATPG) 7-21
7.5.1 Testability 7-21
7.5.2 Fault Model 7-21
7.5.3 Scan Design 7-22
7.5.4 Automatic Test Pattern Generation 7-23
7.5.5 Migration Flow 7-23
7.5.6 TDL'91 7-32

7.6 Copy Protection of Antifuse Architecture 7-33
7.6.1 Antifuses vs. Fuselinks 7-33
7.6.2 Logic Probing or Voltage-Contrast Microscopy 7-33
7.6.3 Proprietary Fuse Addressing and Placement 7-34
7.6.4 FPGA Security Fuses 7-34

8 Application Examples 8-1

8.1 Incorporating a Keypad and Display in an FPGA Design 8-2
8.1.1 Keypad Scanner Design 8-2
8.1.2 Seven-Segment Display Decoder Design 8-4
8.1.3 Keyboard Scanner and Decoder Implementation 8-7

8.2 Implementing a DRAM Controller 8-22
8.2.1 Functional Description 8-22
8.2.2 Memory Arbitration 8-24
8.2.3 Refresh Rate Generator and Timing Control 8-25
8.2.4 Address Multiplexer 8-26
8.2.5 DRAM Controller Design Flow 8-27
8.2.6 Design Flow Using TI ProLogic Software 8-27

8.2.7 ProLogic Design Flow 8-30
8.2.8 Design Flow Using the ABEL Design Software 8-35
8.2.9 Design Implementation in a TPC1010A 8-38
8.3 Implementing CPUs With FPGAs 8-39
8.3.1 Overview of CPU Design 8-39
8.3.2 CPU Implementation 8-43
8.4 Universal Asynchronous Receiver-Transmitter (UART) 8-56
8.4.1 Signal Definitions 8-56
8.4.2 Transmitter Circuitry 8-58
8.4.3 Receiver Circuitry 8-66
8.5 IEEE 1149.1 Boundary Scan Architecture 8-70
8.5.1 Integrated Circuit Level View of 1149.1 8-70
8.5.2 Test Access Port 8-72
8.5.3 Instruction Register 8-76
8.5.4 Bypass Register 8-79
8.5.5 Identification Register 8-79
8.5.6 Boundary Scan Register 8-79
8.5.7 IEEE 1149.1 Boundary Scan Library Components 8-79
8.5.8 IEEE 1149.1 Test Architecture Design Options 8-102
8.6 Pipelined Multiplier 8-114
8.6.1 Mathematical Basis 8-114
8.6.2 Other Methods Investigated 8-115
8.6.3 Design Capture 8-115
8.6.4 Logic Simulation 8-124
8.6.5 Static Timing Analysis 8-126

Glossary G-1

Figures

1-1 Process Technology Progress ... 1-3
1-2 Programmable Array Logic ... 1-4
1-3 Programmable Logic Array ... 1-5
1-4 Generic Array Logic ... 1-6
1-5 Macrocell as an Address Decoder ... 1-7
1-6 A Multilevel Array Architecture ... 1-8
1-7 TI Field Programmable Gate Array ... 1-9
1-8 FPGA Logic Module ... 1-10
1-9 PLD Resources ... 1-11
1-10 Device Architecture ... 1-15
1-11 Architecture Detail ... 1-16
1-12 FPGA Antifuse ... 1-17
1-13 Dedicated Clock Buffer Network ... 1-18
1-14 TPC10 Series Logic Module ... 1-19
1-15 TPC10 Series Clock Distribution ... 1-20
1-16 TPC12 Series Architecture ... 1-21
1-17 TPC12 Clock Distribution ... 1-22
1-18 Logic Function Implementation ... 1-24
2-1 Design Flow ... 2-3
2-2 Counter Example ... 2-5
2-3 Top-Level Schematic ... 2-6
2-4 Workview User Interface ... 2-7
2-5 Example Simulation Command File ... 2-8
2-6 Pre- and Postlayout `trace.poc` Vector Files ... 2-10
2-7 Simulation Waveform ... 2-12
2-8 Workview Screen Layout ... 2-13
2-9 TI-ALS/OrCAD Design Flow ... 2-17
2-10 OrCAD Schematic Diagram ... 2-19
2-11 Simulation Waveforms ... 2-23
2-12 EDIF 2.0.0 Netlist for the RINGCNTR Design ... 2-25
2-13 Sample ADL Netlist ... 2-29
2-14 TI-ALS Valid Design Flow ... 2-32
2-15 TI-ALS Viewlogic Design Flow ... 2-38

2-16 TI-ALS Viewlogic Directory Structure 2-39
2-17 Adding Pins to a Viewlogic Design 2-41
2-18 Pixel Coding 2-44
2-19 Possible Noise Detector Patterns 2-45
2-20 Matrix Numbering 2-45
2-21 Top-Level Schematic 2-46
2-22 Matrix Schematic 2-47
2-23 Mentor/TI-ALS FPGA Design Flow 2-49
2-24 Simulation Command File 2-51
2-25 Prelayout Simulation Output 2-52
2-26 Postlayout Simulation Output 2-54
2-27 Postlayout Delay for Clock-to-ndout 2-56
2-28 Static Timer Analysis Output 2-57
3-1 CAE Environment and TI Action Logic System 3-2
3-2 Schematic Capture and Simulation 3-3
3-3 Pin Assignment and I/O Placement 3-3
3-4 Validation 3-4
3-4 Place and Route 3-4
3-6 Static Timing Analysis 3-5
3-7 Programming 3-5
3-8 Test and Debug 3-6
3-9 TI-ALS Design Flow 3-7
3-10 Workview Main Menu and Export Submenu 3-12
3-11 Export Submenu in the Original `viewdraw.mn8` File 3-13
3-12 Export Submenu in `viewdraw.mn8` File 3-13
3-13 Various Batch Files 3-15
3-14 AP1 Top-Level Schematic 3-18
3-15 NANDBLK Sublevel Schematic 3-18
3-16 ADL Netlist Listing 3-219
3-17 An 8-Bit Full Adder Using Soft Macro FADD8 3-23
3-18 Unit Delay Simulation of 00000001 + 00000001 3-24
3-19 Input Stimuli to Full Adder 3-25
3-20 Postrouting Simulation of 00000001 + 00000001 3-26
3-21 Postrouting Simulation Results Shown on Schematic 3-27
3-22 $CHIP_1$ Including I/O Buffers and Components IN/OUT 3-29
3-23 $CHIP_n$ Including I/O Buffers and Components IN/OUT 3-29
3-24 Viewsim Wirelister Usage 3-29
3-25 Schematic for Multichip Simulation 3-30
3-26 Modified `viewdraw.ini` File 3-31

3-27 Design Used for Timing Analysis 3-32
3-28 Prelayout Versus Postlayout Delays 3-33
3-29 Register-to-Register Delay 3-34
3-30 Asynchronous Loop 3-35
3-31 Four Default Sets 3-35
3-32 Clock Skew 3-41
3-33 Logic Levels 3-44
3-34 Fan-out 3-45
3-35 Single-Level Hard Macro Delay 3-46
3-36 Double-Level Hard Macro Delay 3-47
3-37 Soft Macro Delay 3-48
3-38 Input Buffer Delay 3-49
3-39 Output Buffer Delay 3-50
3-40 Sequential Path Delay 3-51
3-41 Logic Module with Output Track and Long Vertical Track 3-54
3-42 Evaluation Circuit 3-55
3-43 TPC1010 Pinout, 68-Pin Package 3-56
3-44 Example.`ipf` File Shows the Fixed Pins 3-56
3-45 Example `.pin` File Shows All Fixed and Unfixed Pins 3-57
3-46 Location (`.loc`) File Shows the Component Locations 3-58
3-47 TPC1010 Floorplan 3-59
3-48 Placement Information (`.pli`) File 3-60
3-49 Long Vertical Track 3-61
3-50 Path Delays as Analyzed by the Timer 3-62
3-51 TPC1020 Floorplan 3-63
3-52 FPGA Debugging 3-65
3-53 Debug Flow 3-66
3-54 TI-ALS Debug Menu 3-67
3-55 4-Bit Counter TA161 3-68
3-56 TA161 Waveforms 3-69
3-57 Command File `cnt4.deb` 3-70
3-58 Example Outfile `cnt4.res` 3-71
3-59 Example Comparison File `cnt4.cmp` 3-73
3-60 Actionprobe Setup 3-75
3-61 Configuring the Actionprobe 3-76
3-62 Resources Used for 68 PLCC Device 3-77
4-1 FPGA Logic Synthesis Flow with ALES 1 4-4
4-2 State Diagram Sequence Detector 4-5
4-3 Source Files for ALES 1 4-6

4-4 Top-Level Schematic Including I/O Buffers 4-7
4-5 Back Annotation Simulation 4-8
4-6 FPGA Logic Synthesis Flow with Synopsys 4-9
4-7 VHDL Source File, Synopsys Script File 4-10
4-8 Sequence Detector Schematic 4-12
4-9 ALES 1 Design Flow for PC 4-15
4-10 PALASM 2 Source File 4-18
4-11 Logic Synthesis Design Flow 4-20
4-12 Counter Example 4-22
4-13 PLDesigner-XL Architecture 4-25
4-14 Flow Diagram Using PLDesigner-XL to Target a TI FPGA 4-29
4-15 Synopsys to FPGA Migration Flow 4-31
4-16 Synopsys File Hierarchy 4-32
4-17 Synopsys Compilation Flow 4-35
4-18 DUMMY File 4-37
4-19 Include Scripts Which Configure the Compilation of the Top and DUMMY Files 4-39
4-20 Include File That Sets Up Correct Library for Creating the Schematics 4-40
4-21 ADL Correction Flow 4-42
4-22 Conversion Errors 4-43
4-23 Layout_fpga Script 4-46
4-24 Conversion Program 4-47
4-25 Block Diagram 4-51
4-26 Timer State Diagram 4-52
4-27 Time_state_machine 4-53
4-28 Design Flow 4-55
4-29 Test_timer.v File 4-56
4-30 Verilog Options File 4-57
4-31 Example `.synopsys` File 4-58
4-32 Synopsys Control Script for Timer 4-59
4-33 Top-Level Synthesis Schematic 4-60
4-34 TPC10 Series Implementation of TIME_STATE_MACHINE 4-61
4-35 Conditional Statement to Invoke MDI-Injector for Postlayout Simulations 4-62
4-36 Prelayout Simulation Result 4-64
4-37 Prelayout Simulation, CLK to HOURS-OUT Delay of 3 ns 4-65
4-38 $Monitor Output 4-66
4-39 Simulation Switches 4-67
4-40 Postlayout Simulation, CLK to HOURS-OUT Delay of 69 ns 4-68
4-41 $Monitor Output for Postlayout Simulation 4-69
4-42 TI-ALS Timer Results 4-71

4-43 Simple State Transition 4-74
4-44 Equivalent Statechart 4-75
5-1 Factors/Results Screen 5-3
5-2 Results for a Single Logic Family with Maximum Constraints Enforced 5-5
5-3 Results for a Single Logic Family with Maximum Constraints Relaxed 5-6
5-4 Results for Multiple Logic Families with Maximum Constraints Relaxed 5-7
5-5 The proLogic Source File 5-8
5-6 ProLogic-Generated `cnt41.pds` File 5-11
5-7 Example Test Result File (`cnt41.tst`) 5-12
5-8 Symbol Created in Viewdraw 5-14
5-9 TI-ALS Timer Longest Delay Paths 5-16
5-10 TI-ALS Timer Delay Elements 5-16
5-11 Source File for 5-Bit Loadable and Cascadable Counter Using ProLogic 5-18
5-12 Mapping Internal Three-States to Combinational Equivalents 5-22
5-13 Summary of Remapping the Targeting 10 Series 5-23
5-14 Summary of Optimizing the Targeting 12 Series 5-24
5-15 Summary File Targeting 10 Series 5-25
6-1 Minimum Production Quantity for Migration 6-3
6-2 Asynchronous 3-Bit Counter 6-5
6-3 Synchronous 3-Bit Counter 6-6
6-4 Asynchronous 3-Bit Counter With Scan Chain 6-7
6-5 Synchronous 3-Bit Counter With Scan Chain 6-7
6-6 FPGA-to-ASIC Migration Flow 6-9
6-7 TI FPGA-to-ASIC Migration 6-12
6-8 Netlist Translation Flow 6-13
6-9 Scan Cell and Scan Path 6-15
6-10 Sequence Detector State Diagram 6-16
6-11 Sequence Detector FPGA Schematic 6-16
6-12 ADL Netlist 6-17
6-13 TGC100 Schematic Without Test Scan 6-17
6-14 TGC100 Schematic With Test Scan 6-18
6-15 TI TDL Format 6-19
6-16 Logic Synthesis Migration Flow 6-20
6-17 VHDL Code and Script File 6-21
6-18 TI Migration Flow 6-23
6-19 Example of Circuitry Used as Alternative to Replacing Flip-Flops 6-24
6-20 Example of Solution to Gated Clock 6-25
6-21 Example of Poor Design Practice Causing Possible Setup Violations 6-25
6-22 Example of Solution to Using Both Edges of Clock 6-26

6-23 Example of Solution for Uncontrollable Asynchronous Pins 6-27
6-24 Example of Q OUT Used to Clock Flip-Flop 6-28
7-1 FPGA Per-Unit Programming Time 7-3
7-2 FPGA Programming Cycle Time 7-4
7-3 Activator 2 Programmer Throughput 7-5
7-4 68-/84-Pin PLCC Defect-Level Leads 7-6
7-5 Defect Probability and PPM Level 7-7
7-6 Blank vs. TI-Programmed FPGA Process Steps 7-9
7-7 TI P-FPGA Factory Process Flow 7-10
7-8 Antifuse Integrity Test 7 and Integrity Test 8 7-17
7-9 Stuck-at-Fault Model 7-21
7-10 Multiplexed Flip-Flop Scan Methodology 7-22
7-11 FPGA-to-ASIC Design Flow 7-24
7-12 Synopsys Script for TPC10 to TGC100 Translation 7-25
7-13 Script for Scan Insertion Using the Synopsys Test Compiler 7-26
7-14 TPC10 Series Timer State Machine Schematic 7-26
7-15 TGC100 Timer State Machine Schematic With Scan 7-27
7-16 Scan Path Report Generated by the Test Compiler 7-27
7-17 Timer Core Block Showing the Scan Path 7-28
7-18 Synopsys Script to Generate TDL'91 Test Vectors 7-29
7-19 Test Coverage Report 7-30
7-20 TDL'91 Scan Circuitry Test Program (`timer_comp_schk.tdl`) 7-31
8-1 Circuit Schematic of the Keypad Scanner 8-3
8-2 Timing Diagram of the Keypad Scanning Circuit 8-3
8-3 LED Seven-Segment Display 8-6
8-4 Obtaining Logic Functions From the TI FPGA Logic Module 8-8
8-5 Circuit Reduction Using Input Inversions 8-9
8-6 Circuit Schematic for Generating the Intermediate Terms 8-12
8-7 Circuit Schematic for the Keypad Decoder 8-13
8-8 Route Programming 8-19
8-9 Pass Transistors Used for Programming 8-20
8-10 TI-ALS `.vld` File 8-20
8-11 DRAM Controller Block Diagram 8-23
8-12 Memory Arbitration Unit Block Diagram 8-24
8-13 Refresh Rate Generator and Timing Control Block Diagram 8-25
8-14 Address Multiplexer Block Diagram 8-26
8-15 The `.pld` Source File 8-27
8-16 Design Flow Using TI ProLogic Software 8-29
8-17 State Machine Description 8-30

8-18 Example Runs of JEDEC Compilation and Simulation Run 8-31
8-19 The `.tst` File (Simulation Result File) 8-31
8-20 The `.pds` File for Synthesis and Optimization 8-32
8-21 Viewlogic State Machine Simulation Results 8-33
8-22 Viewgen-Created Schematic of Block 8-34
8-23 Integration of proLogic Block With Schematic Blocks 8-35
8-24 ABEL File for State Machine Design 8-36
8-25 ABEL Design Flow 8-37
8-26 CPU Block Diagram 8-40
8-27 Hardware Implementation of Control Signal Latching Address Register 8-45
8-28 Timing Diagram for Read Cycle 8-53
8-29 Top-Level Schematic 8-57
8-30 Receiver and Transmitter Sections With Arbitration 8-59
8-31 UART Transmit and Receive Cycles 8-60
8-32 Receiver Circuit Simulation 8-61
8-33 Transmitter Simulation 8-62
8-34 Transmitter Circuit Blocks 8-63
8-35 Transmit Buffer (TXBUF) 8-64
8-36 Transmit Controller (TXCNTBF) 8-65
8-37 Serial Transmit Register 8-66
8-38 Receiver Circuit Blocks 8-66
8-39 Receiver Buffer 8-67
8-40 Simulation Command File 8-68
8-41 IEEE 1149.1 Architecture 8-69
8-42 1149.1 TAP State Diagram 8-73
8-43 Test Access Port (TAP) 8-70
8-44 TAP Block Diagram 8-82
8-45 TAP Instruction Register Scan Timing Diagram 8-84
8-46 TAP Data-Register Scan Timing Diagram 8-85
8-47 TAP Test Logic Reset Timing Diagram 8-86
8-48 Unidirectional Boundary Cell 1 8-87
8-49 Boundary Register Using UBCELL1 8-88
8-50 Unidirectional Boundary Cell 2 8-88
8-51 Boundary Register Using UBCELL2 8-90
8-52 Bidirectional Boundary Cell 1 8-90
8-53 Boundary Scan Register Using BBCELL1 8-92
8-54 Bidirectional Boundary Cell 2 8-92
8-55 Boundary Register Using BBCELL2 8-94
8-56 Bypass Register (BYPREG) 8-94

8-57 Instruction Register Cell 1 (IRCELL1) 8-95
8-58 Instruction Register Cell 2 (IRCELL2) 8-96
8-59 Example Instruction Register 8-98
8-60 Test Data Output Cell (TDOCELL) 8-100
8-61 Identification Register 1 Macro (IDREG1) 8-101
8-62 Identification Register 2 Macro (IDREG2) 8-101
8-63 Example 1149.1 Test Architecture 8-103
8-64 Example 1149.1 Test Architecture Using TIM1 8-107
8-65 Example 1149.1 Test Architecture Using TIM2 8-110
8-66 Top-Level Schematic 8-116
8-67 Multiplier File Schematic Symbol 8-117
8-68 Multiplier File Block Description 8-118
8-69 Multiplicand File Schematic Symbol 8-118
8-70 Multiplier/Adder Block Schematic Symbol 8-119
8-71 Sublevel Multiplier/Adder Schematic 8-122
8-72 Register Block Schematic Symbol 8-123
8-73 Register Block PASLASM Source File 8-123
8-74 R12 Block Schematic Symbol 8-123
8-75 Logic Simulation Waveform (1 of 2) 8-124
8-76 Logic Simulation Waveform (2 of 2) 8-125
8-77 Simulation Log File 8-125

Tables

3-1 Device Package Pins and Assignments 3-77
3-2 Criticality Levels and Validator Errors and Warnings 3-80
4-1 Module Count for the Different FPGAs 4-33
6-1 Typical Alternative Costs 6-2
6-2 3-Bit Counter State Table 6-6
6-3 Fault Report for Counters 6-8
7-1 TPC10 Series Binning Path Pin Assignments 7-20
8-1 Boolean Equations for the Intermediate State 8-4
8-2 Truth Table 8-5
8-3 Boolean Equations for the Design of the Display Decoder 8-6
8-4 Boolean Equations Based on Intermediate Terms 8-10
8-5 Boolean Equations With the Additional Inversions 8-11
8-6 First Longest Path to All Endpins 8-16
8-7 First Longest Path to FF4:D (Rising) (Rank: 0) 8-17
8-8 First Longest Path to All Endpins 8-18
8-9 List of Instructions Supported by v3.2 CPU 8-41
8-10 Sequence of Nine Micro-Instructions Implementing the ADD Instruction 8-42
8-11 Logic Equation for Control Signal #1 8-45
8-12 Component Delay Through Data Register and ALU 8-49
8-13 TAP State Table 8-83
8-14 TAP Outputs During Test Logic Reset State 8-86
8-15 Unidirectional Boundary Cell 1 Truth Table 8-87
8-16 Unidirectional Boundary Cell 2 Truth Table 8-89
8-17 Bidirectional Boundary Cell 1 Truth Table 8-91
8-18 Bidirectional Boundary Cell 2 Truth Table 8-93
8-19 Bypass Register (BYPREG) Truth Table 8-95
8-20 Instruction-Register Cell 1 (IRCELL1) Truth Table 8-96
8-21 Instruction-Register Cell 2 (IRCELL2) Truth Table 8-97
8-22 Instruction-Register Decode Truth Table 8-99
8-23 Test Data Output Cell (TDOCELL) Truth Table 8-100
8-24 Identification Register 1 Macro (IDREG1) Truth Table 8-101
8-25 Identification Register 2 Macro (IDREG2) Truth Table 8-102
8-26 Module Count Look-Up Table 8-106

8-27 TIM1 Functional Description 8-109
8-28 TIM2 Functional Description 8-113
8-29 Example Multiplication 8-114
8-30 Internal Frequency Measurement 8-126
8-31 Clock-to-Output Delay 8-127
8-32 t(datapath) Delays 8-128
8-33 t(clock) Delays 8-128

Chapter 1

Introduction

This chapter explores the development of the field programmable gate array (FPGA) and introduces the FPGA architecture, applications, and design benefits.

1.1 Field Programmable Gate Arrays—Revolutionizing Logic Design†

If the automotive industry had made the same progress as the electronics industry in the last two decades, a typical car today would be the size and the weight of a box of matches, could be driven at several thousand kilometers per hour, use one liter of gasoline per 1000 km and cost perhaps $100. This comparison may appear exaggerated, but it shows key forces of the development in the electronics industry: miniaturization, performance, power consumption, and cost.

The microelectronics revolution was enabled because significant improvements were achieved in many areas: silicon process technologies, semiconductor components, and design tools are some of the most important. Obviously, the invention of the microprocessor, the availability of affordable, high-density memory chips, and the advancements in logic products such as general purpose logic products (GPL), programmable logic devices (PLDs) and application specific integrated circuits (ASICs), have played the key roles. Section 1.1, including the following subsections, discusses the issues of logic implementation by investigating the architecture, applications, benefits, and drawbacks of the different options offered by GPL, PLDs, FPGAs, and ASICs.

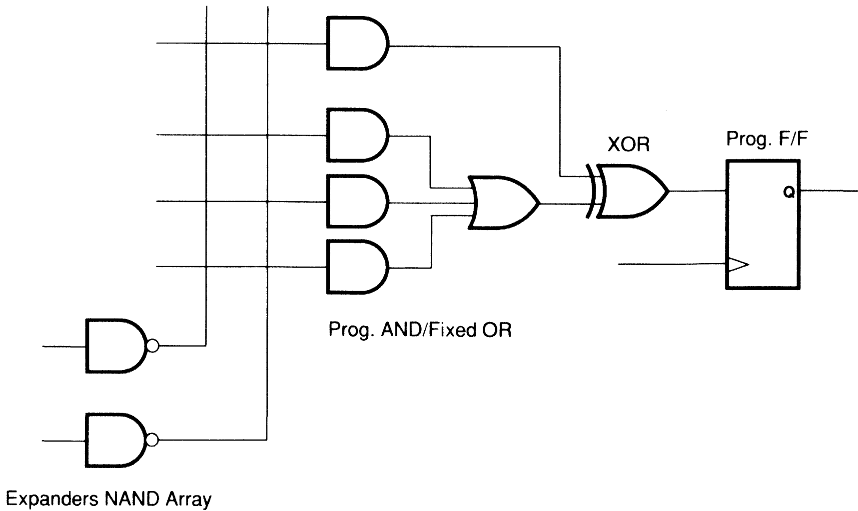
1.1.1 General Purpose Logic

In the 1970s, when transistor-transistor logic (TTL) integrated circuits (ICs) were introduced to the market, they were simply the ultimate product for digital designers. TTLs were used in nearly every application. Over the years, general purpose logic products have been built in all kinds of process technologies: standard TTL, low power Schottky (LS), advanced low power Schottky (ALS), standard Schottky (S), advanced standard Schottky (AS), fast (F), HCMOS, advanced CMOS logic (ACL), BiCMOS (BCT) and advanced BiCMOS technology (ABT).

Figure 1-1 illustrates, as an example, the evolution of the popular octal bus transceiver 245. Compared with the bipolar low power Schottky LS245, the advanced BiCMOS technology ABT245 is 3x faster, the drive capability is 2.6x higher but the power consumption is reduced by a factor of 18.5. This example demonstrates the great advancements which may be achieved in process technology.

† Contributed by Dung Tu, Technical Marketing, Texas Instruments Deutschland GmbH.

Figure 1-1. Process Technology Progress

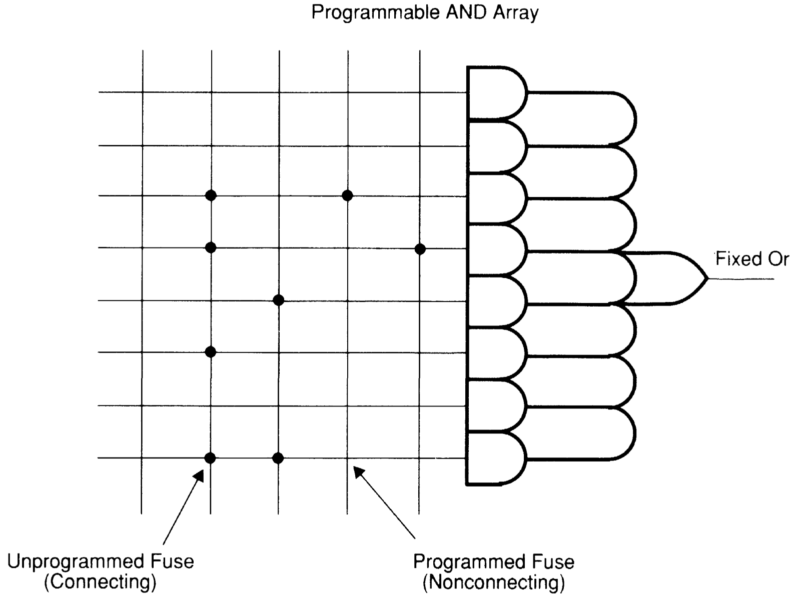


Traditionally, GPL products can be classified into four groups: gates, flip-flops, medium/large scale integration (MSI/LSI), and bus functions. With the exception of the bus functions (where GPL products cannot be surpassed due to their excellent drive capability, speed, and cost), other GPL products do not provide the required complexity and flexibility for today's new designs, so they are being replaced by programmable logic devices and gate arrays. New GPL products will be mainly for wide bus functions (16 to 20 bits) using advanced BiCMOS technology.

1.1.2 Programmable Logic Devices

At the end of the 1970s, the programmable array logic (PAL®) device was introduced to the market. This product has a programmable AND array which feeds into a fixed OR array (Figure 1-2). The first PAL circuit was the 16L8, which was a combinational, 20-pin PAL device with eight outputs in bipolar technology. Instead of using several GPL logic chips like LS00, LS04, etc., to build, for example, an address decoder, you could put all these gate functions into one PAL device. This gain in complexity over GPL is one advantage of PAL circuits.

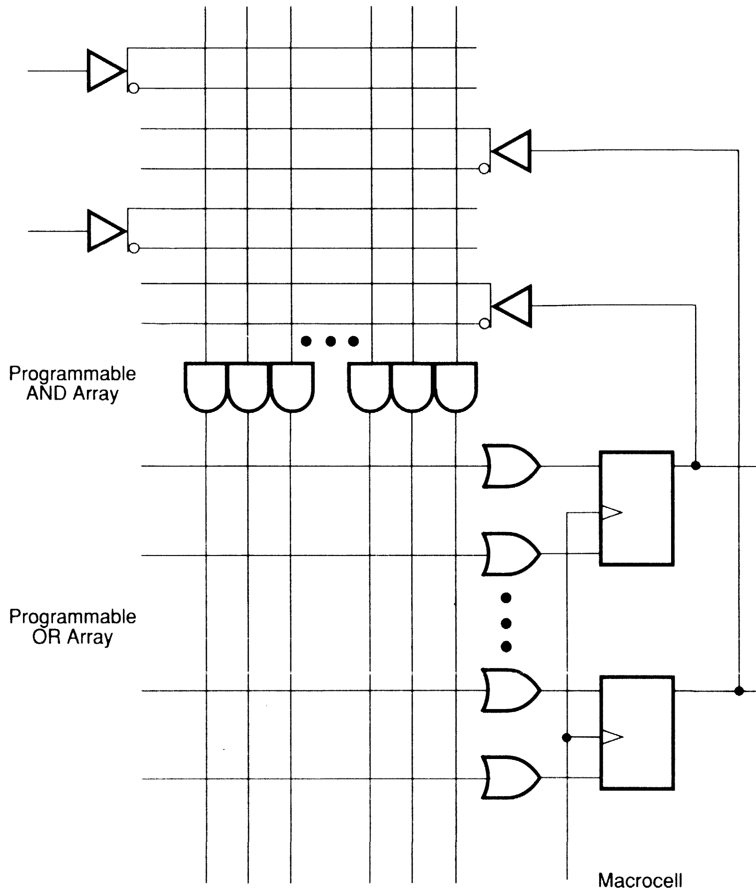
Figure 1-2. Programmable Array Logic



However, a real innovation was the fact that PAL devices are user-programmable. Using PAL circuits, you could for the first time design and program an application-specific chip within a very short timespan. Determined by the speed improvements of the microprocessors, where the clock rate has increased from 4.7 MHz in the first IBM PC to 33 MHz in today's PC-386 (or even 50 MHz in some workstations), PAL devices are becoming increasingly faster. Today, 5-ns PAL devices are clearly the product of choice for glue logic when high speed is required.

Another PLD product called a programmable logic array (PLA) was also taken to market at the same time as the PAL device. A PLA has a programmable AND array which feeds into—in contrast to a PAL circuit—a programmable OR array (Figure 1-3).

Figure 1-3. Programmable Logic Array

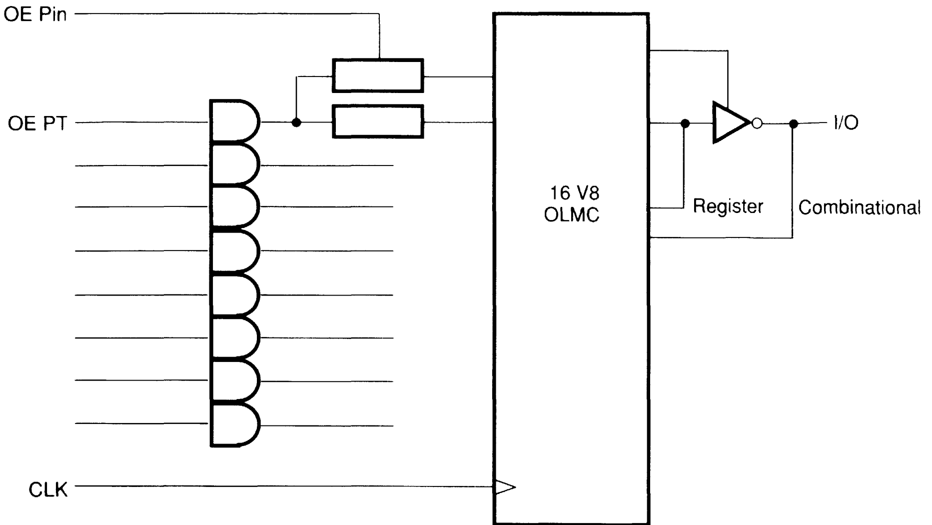


In terms of architecture, a PLA provides more flexibility than a PAL device because the number of product terms are not fixed, but programmable. There are products with PLA architecture, such as sequencers, which allow you to program up to 100 product terms to one output. This feature makes PLAs suitable for complex state machine applications where state decoding requires many product terms.

While speed and *quiet* signal edges are advantages of bipolar PAL devices, their disadvantages are clearly high power consumption and unconfigurable

output cells. The outputs of a 16L8 can be used only for combinational logic and cannot be configured as flip-flops. The opposite is true for a 16R8.

Figure 1-4. Generic Array Logic

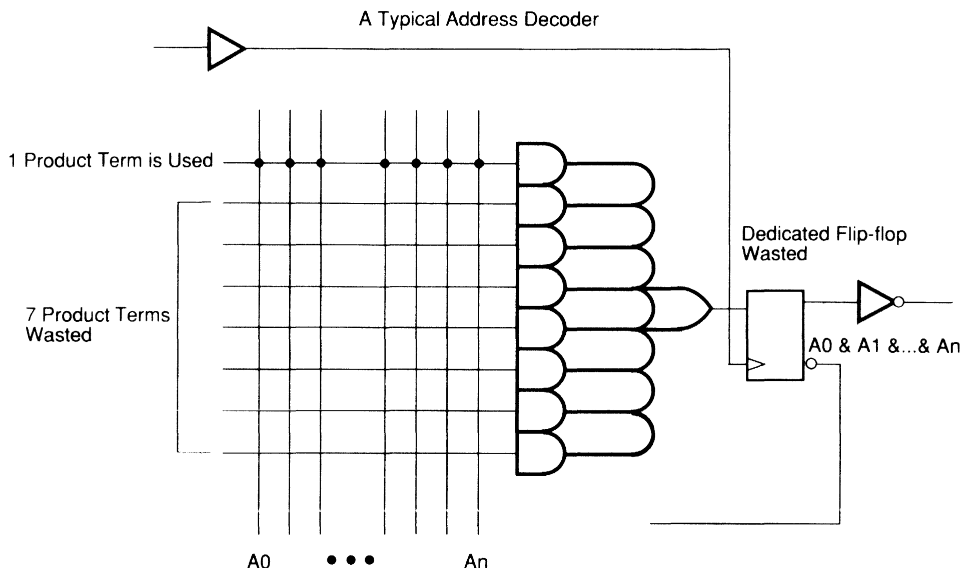


Generic array logic (GAL™), shown in Figure 1-4, has output macrocells which can be configured as combinational or registered logic. Thus, a single GAL like the 16V8 can replace 24 different PAL types. The GALs are built in CMOS—this reduces the power consumption compared to bipolar PAL devices. However, one problem of GALs is the fast edge rate which can cause a large amount of noise on the board. Therefore, GALs cannot replace bipolar PAL devices in old designs.

While GALs have the same pin counts as PAL devices (20 and 24), erasable programmable logic devices (EPLD) provide pin counts from 20 to 68 pins. The EPLD register is programmable as D, T, JK or SR flip-flop, with up to four independent clocks, half the power consumption of GALs, zero standby power, higher pin counts, and packed with an attractive, low cost and easy to use development system. The EPLDs are not fast, but, due to the low power consumption, they are very successful in telecommunications and industrial applications.The largest EPLD, the EP1810, which is equivalent to four 16V8s in one 68-pin package, is the first successful approach to make a large, complex PAL device. However, EPLDs still have the typical programmable AND/fixed OR arrays. The problem of this architecture is the low-gate utilization.

Figure 1-5 shows a macrocell which is used as an address decoder. The macrocell has eight product terms but only one of them is used. The macrocell has a dedicated flip-flop which is bypassed. However, when an application really needs more than eight product terms, you must use feedback loops which slow down the design significantly.

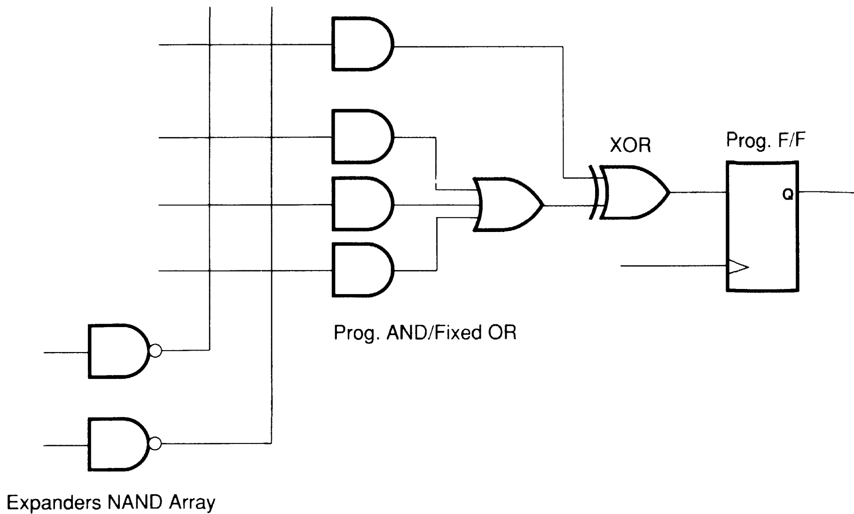
Figure 1-5. Macrocell as an Address Decoder



To overcome these problems, new products have been taken to market in the past two years. These products are different in detail, but in general they can be considered as an approach to combine both PAL devices and PLAs in one product which can be called *multilevel arrays* or complex PLDs.

Figure 1-6 depicts one such architecture. The macrocell has only three product terms instead of eight. When more than three are required, additional NAND gates from a so-called expander array can be switched to the macrocell. The macrocell itself has an AND/OR structure so that it can emulate a PAL circuit. However, because the output of the OR gate is connected to a XOR, the combination AND/OR/XOR can be configured as a NAND. Thus, the NAND expander term plus the (as NAND) configurable macrocell build a NAND/NAND structure which is equivalent to an AND/OR. Because both the expander and macrocell arrays are programmable, this architecture also emulates the PLA architecture.

Figure 1-6. A Multilevel Array Architecture

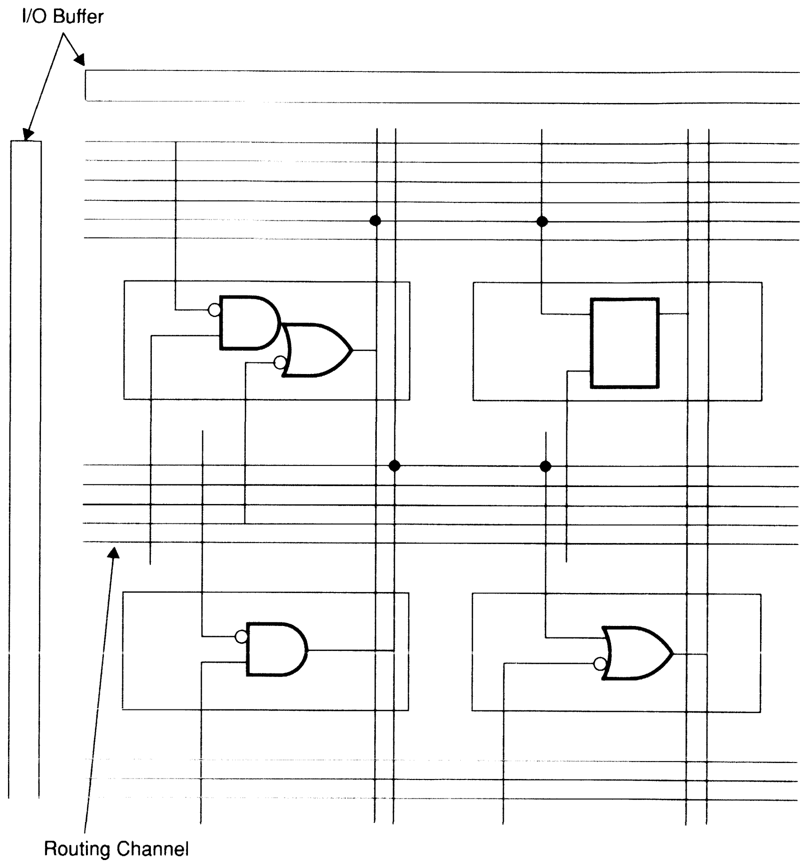


1.1.3 Field Programmable Gate Arrays

Field programmable gate arrays (FPGAs) are the most exciting logic products today. The driving forces for the development of field programmable gate arrays are basically the same as for the whole electronics industry but time-to-market and density are apparently the two key elements. Compared to classic gate arrays, the FPGA chip price is higher. You get, however, a user-programmable chip with a density which is 10–100x higher than that of a PAL device. You can also make an FPGA within a week without paying any nonrecurring engineering (NRE) charges. This significantly reduces the risk for new designs which can ultimately result in considerable cost reduction. For low volume, FPGAs are in fact very cost-effective.

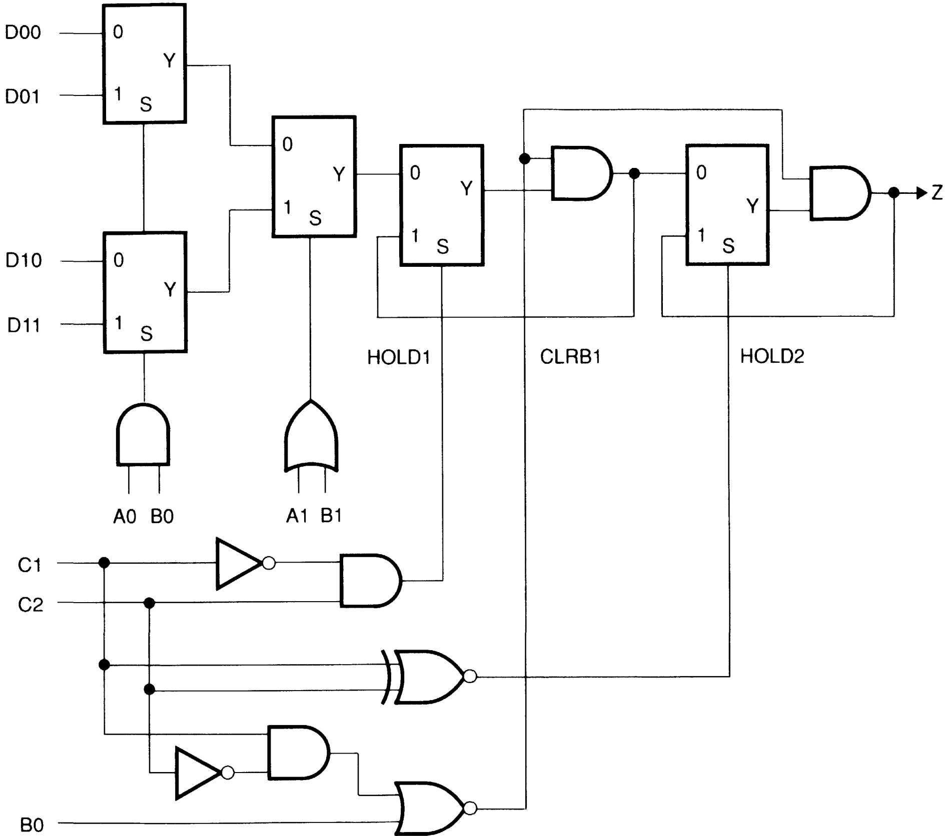
Figure 1-7 shows the architectures of the TI FPGA. In the opinion of the TI FPGA team, this particular FPGA is the best of several FPGA architectures which utilize different interconnect fusing structures (antifuses, SRAM cells, EPROM, or EEPROM) and different-sized logic elements (ranging in size from large—30–40 gates—to small—1 NAND gate).

Figure 1-7. TI Field Programmable Gate Array



The FPGA from TI has a channeled gate array-like architecture with logic modules as building blocks. Nonvolatile antifuses and routing channels are used for interconnections. The logic element in the TI FPGA architecture should be considered to be medium-sized in that is can implement up to approximately 8–10 gates per module.

Figure 1-8. FPGA Logic Module

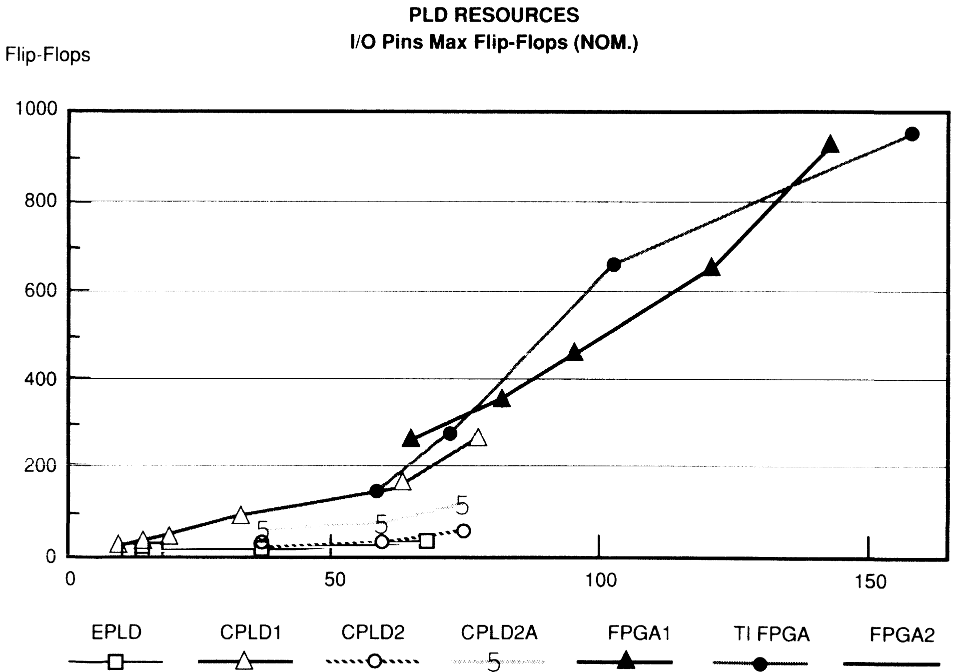


A large building block can be very efficient for applications which fit into the architecture but it can suffer otherwise from I/O limitation problems and is more difficult to migrate to ASIC. On the other hand, a small building block leads to a high nominal gate count and is most *gate array-like* but requires increased routing resources; if the routing resources are not available the

utilization is very low and place and route can become a significant problem. Today, channeled architecture is the compromise which has the highest utilization and the best routability.

Using gate count to quantify the density of programmable logic devices is an important issue. Depending on the gate utilization and the applications, the number of usable gates can differ significantly from the nominal gate count. In fact, the total number of usable flip-flops and I/O pins, which is shown in Figure 1-9, can be used to get a rough idea of the real resources of a PLD product. Complex PLDs can provide a maximum of approximately 300 flip-flops while FPGAs can implement up to approximately 1000 flip-flops.

Figure 1-9. PLD Resources



The FPGA2 architecture consists of small logic segments and is aimed at high nominal gate count to prototype gate arrays. However, the utilization is less than 30 percent. To be successful, the problem with routing resources and place and route must be solved. The TI FPGA can provide a complete product spectrum from 1K to approximately 8–10K gates.

Good design tools are very essential for FPGAs. A development system for FPGA consists normally of two parts: a universal CAD environment for design entry/simulation and a product-specific tool for place and route, extract timing, and programming.

Place and route is clearly the most important part of the tool. It can be run one hundred percent automatically and be completed within 30 minutes, or it can run for many hours and route only part of the design (like some other tools). Due to high complexity (the smallest FPGA can replace ten PLDs), back annotation simulation after place and route is very helpful to verify the timing. Because a design normally needs several modifications before it is functional, a long cycle time for place and route and back annotation simulation has a very negative impact on time-to-market. This is why the TI FPGA software was designed to perform without manual intervention.

The support of logic synthesis tools is also important for FPGAs. Logic synthesis improves not only your productivity but can also optimize the speed of the design. It allows a technology-independent design methodology and is useful for migrating from FPGA to ASIC.

1.1.4 Application Specific Integrated Circuits

By the year 2000, 40 percent of total semiconductors in use worldwide (including microprocessors and memory chips) are forecast to be ASICs. There is no doubt that gate arrays and standard cells have become an essential tool for making electronics systems. Without ASICs, products like PC chip sets, ISDN, digital TV, video games, etc., would not be feasible. ASICs are used for applications requiring several hundred gates as well as 200K gates. If the volume is high enough, CMOS gate arrays are an unbeatable solution both in terms of performance and cost. At the high end, BiCMOS gate arrays (for up to 150K gates) with true sandwich CMOS-bipolar-CMOS cells, which allow 300 ps of internal gate delay, provide fast compiler memories/datapaths (3–6 ns SRAM) and more than 300 I/Os for high-performance system solutions on one or two chips. However, traditional gate-level design for such complexity would be very inefficient. Logic synthesis with behavioral models or VHDL is the tool of choice.

You have never had a greater choice of possible solutions for a specific task as you have today. Each technology has specific benefits and drawbacks. GPL provides the best bus interface functions, while bipolar PLDs give the highest speed for glue logic. CMOS GALs are cost efficient, and complex multilevel-array PLDs can replace several PAL devices. However, the future belongs to FPGAs and classic gate arrays. Because continuous advances in process technologies can be expected, FPGAs with combined advantages of channeled triple-level metal architecture and sea-of-gates in submicron CMOS/BiCMOS technology will be available in this decade.

Today, because the designer careabouts remain basically the same, the following requirements list can be useful in the decision process to select the right logic product for a particular application:

Requirements List:

- Density
  - Total available gates
  - Gate utilization
  - Total available flip-flops
  - Maximum available I/O pins
  - Available packages
  - User-defined pinout/package
  - RAM/ROM on-chip
- Performance
  - Toggle rate
  - System speed
  - Gate delay
  - Wiring delay
  - I/O delay
  - Clock skew
  - Wiring skew
  - Reconfigurability/Nonvolatility
  - Noise, simultaneous switching
- Power consumption
  - Bipolar/CMOS/BiCMOS
  - Drive capability
  - Stand-by power/Dynamic power
- Cost
  - Chip cost
  - Multisources
  - Price/performance of development system
  - NRE
  - Migration to ASIC
  - Hardwired option
  - Preprogramming service
  - Inventory
  - Multisources
  - One-chip /two-chip solution
  - Design protection

Requirements List (Continued):

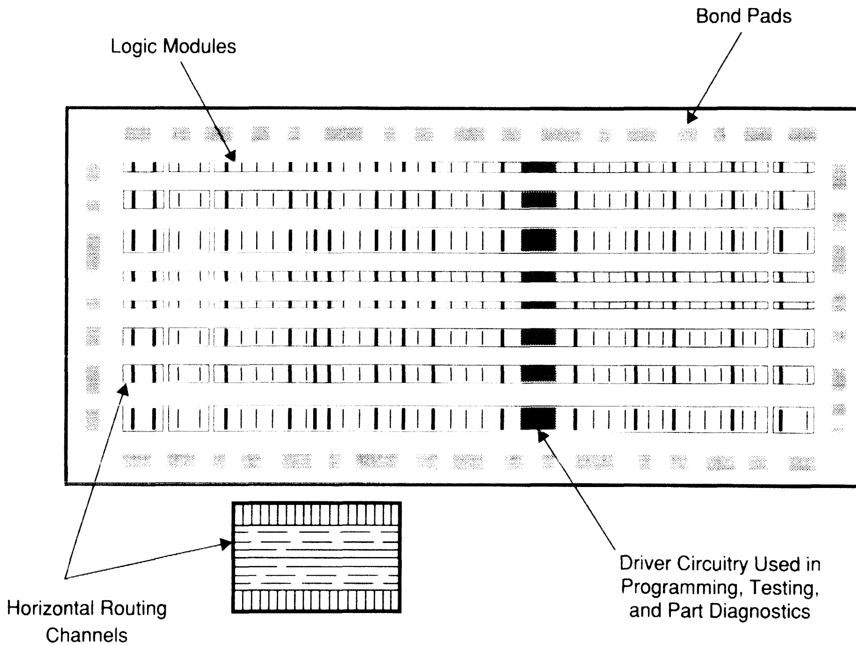
- Time-to-market
  - User-programmable
  - Good automatic place and route tool
  - Ease of use
  - Support popular CAD platforms, good simulator
  - Support for logic synthesis tool
  - Debug/testability
  - Multisources
  - Availability/delivery
  - Technical support
  - Training
  - Design service

1.2 FPGA Device Architecture†

1.2.1 Introduction

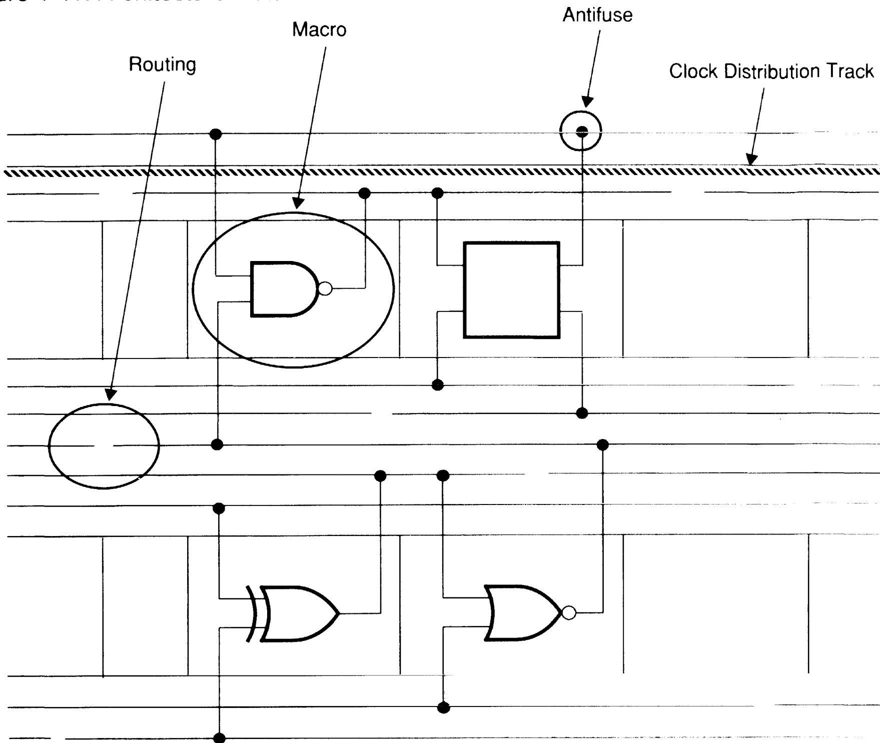
To develop a good idea of the capabilities of a field programmable gate array (FPGA), a knowledge of its architectural elements is needed; see Figure 1-10 and Figure 1-11. The TI FPGAs have several characteristic features which define their function. These basic features are the antifuse, the logic module, the clock distribution network, routing channels, and diagnostic circuits.

Figure 1-10. Device Architecture



† Contributed by Joel S. Lason, P.E., FPGA Applications, Texas Instruments Incorporated.

Figure 1-11. Architecture Detail



The two TI FPGA families are designated by the nomenclature TPC10 and TPC12. Both families share many architectural features, but the newer TPC12 family has some enhancements. A general discussion of the elements common to both devices is presented in Section 1.2.2. Then each family's unique components are covered in Section 1.2.3 and Section 1.2.4. And finally, an example of the way that a logic module is used to implement functions is explained in Section 1.2.5.

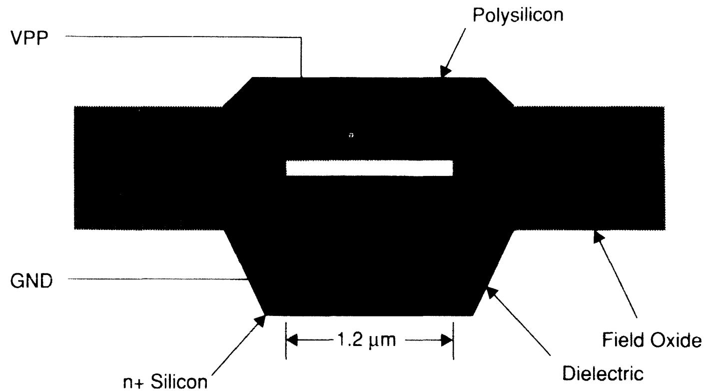
1.2.2 Architectural Overview

1.2.2.1 The Antifuse

The element which provides programmability to the device is the antifuse, which is shown in Figure 1-12. An antifuse is a normally open device which becomes conductive when a high-voltage pulse is applied to it. This is opposite to the action of a fuse which becomes nonconductive when a large

voltage or current is applied to it, as in bipolar PAL devices. As Figure 1-12 shows, the antifuse consists of two conducting layers separated by a thin isolation layer. Another important feature is the small size which is up to 19x smaller than other types of programmable elements.

Figure 1-12. FPGA Antifuse



Metal routing tracks run horizontally and vertically over the die, and at each intersection between these metal tracks there is an antifuse. The unprogrammed resistance of the antifuse is greater than 100 MΩ and the programmed resistance is approximately 500 Ω.

1.2.2.2 Routing

As Figure 1-10 and Figure 1-11 show, the FPGA architecture is analogous to that of a channeled gate array with routing tracks running horizontally between the logic modules and vertically over them. These tracks interconnect the macro functions implemented in the logic modules. The number of routing tracks per channel is family-dependent and the segmentation of these tracks varies from device to device. Segmentation refers to the number of logic modules that a track spans before it has a break in it.

1.2.2.3 Logic Module

A prominent feature of the architecture is the rectangular array of logic module elements which contain generic logic circuits. These circuits can be programmed to perform a variety of functions. Each logic module is multiplexer-based. A TPC10 series device has only one type of logic module while a TPC12 series device has both the TPC10 series module and an

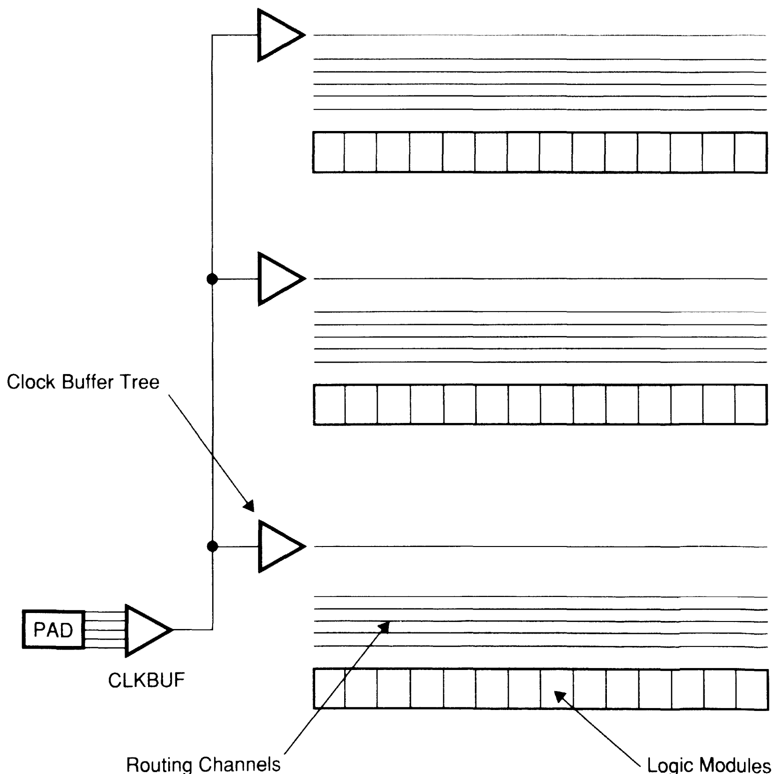
enhanced sequential module. Sequential TPC10 series functions are implemented using two combinational modules.

The macro functions implemented in the logic modules are generated in several ways. They come from TI-supplied hard and soft macro libraries, or you can create them.

1.2.2.4 Clock Distribution

High drive requirements for the clock signal are met by a dedicated clock buffer network which is shown in Figure 1-13. This network consists of an assigned input pin and row buffers to provide additional drive for the large fan-out requirements of the clock signal. The network can connect to any logic module but only to clock and gated inputs of sequential macro functions.

Figure 1-13. Dedicated Clock Buffer Network



1.2.2.5 Diagnostics

Special circuitry is built into the device for diagnostics. This circuitry consists of shift registers which allow you to address and probe any internal node of the programmed device. Four special I/O pins are provided for this function. These pins are multiplexed so that they can be configured as diagnostic or normal I/O pins and are controlled by the debug function of the TI Action Logic™ System (TI-ALS). This function allows the internal nodes to be probed while the device is operating in an actual circuit board.

1.2.2.6 Inputs/Outputs

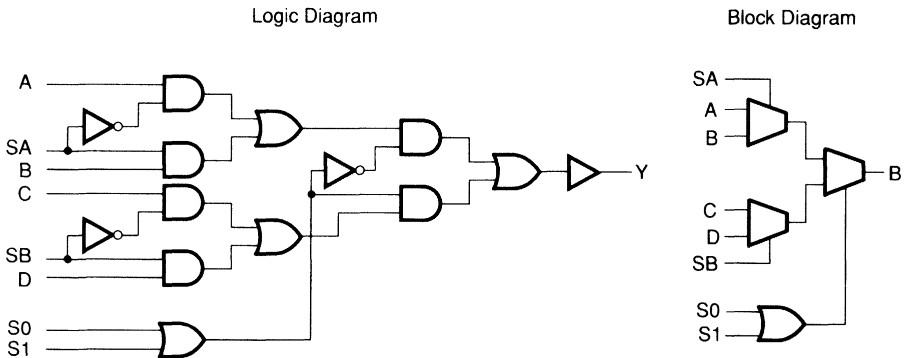
The I/O count depends on the device and the packaging of the device. The I/O function can be programmed as either input, output, 3-state, or bidirectional.

1.2.3 TPC10 Series Specific Features

1.2.3.1 Logic Module

The TPC10 series logic module is shown in Figure 1-14. Every logic module has eight inputs and one output and emulates the function of three 2-input multiplexers and one OR gate. Flip-flops are created by connecting a couple of modules in the appropriate circuit configuration. The medium module size allows efficient mapping of logic functions and thus high device utilization.

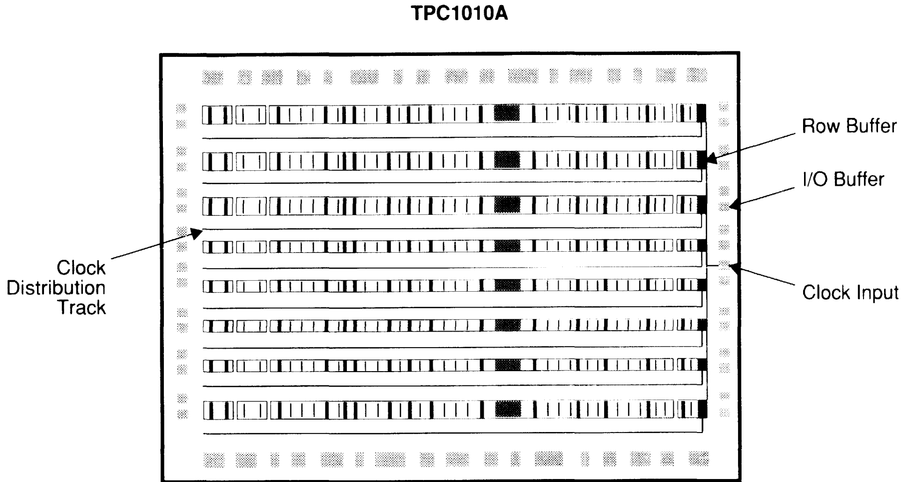
Figure 1-14. TPC10 Series Logic Module



1.2.3.2 Clock Distribution

A detailed diagram of clock distribution on the TPC10 series device is shown in Figure 1-15. The TPC10 series parts have a single dedicated clock network that can hold clock skew to less than 3 ns. There are no load restrictions on the clock network, and the row buffers help to reduce overall loading on the clock network input pin.

Figure 1-15. TPC10 Series Clock Distribution



1.2.3.3 Routing

The TPC10 series devices have 25 horizontal routing tracks per channel. Twenty-two are for logic, one is for clock, one is for V_{CC}, and one is for ground. In addition, there are 13 vertical routing tracks. Tracks that extend more than a third of a row or column are referred to as long vertical tracks. These tracks can impose an additional speed penalty on the design.

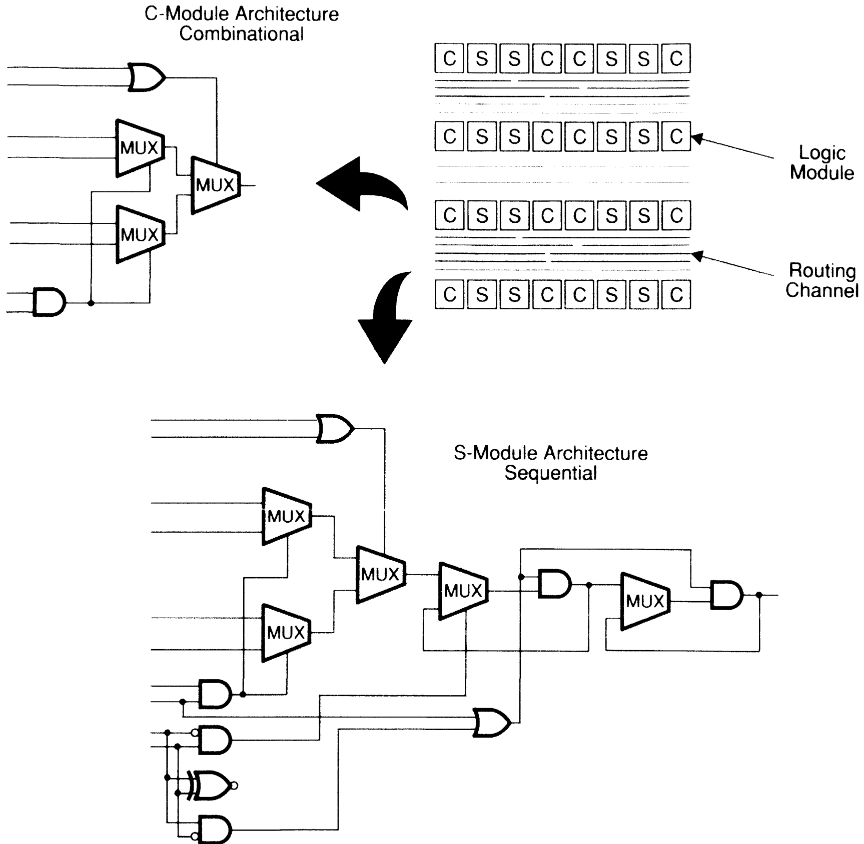
1.2.4 TPC12 Series Specific Features

1.2.4.1 Logic Module

Additional sequential capability has been added to the TPC12 family device in the form of a sequential module as shown in Figure 1-16. On a TPC12 series die there are as many as 624 of these enhanced modules. Each sequential module consists of the TPC10 series OR gate-multiplexer structure plus a flip-flop that is fed by the combinational output. There are

also as many as 608 combinational modules of the type that exist on the TPC10 series die. The TPC12 series combinational module also has an extra AND gate. Each sequential module is a complete flip-flop, and a flip-flop can be made from two combinational modules so that total flip-flop count can be 928 flip-flops.

Figure 1-16. TPC12 Series Architecture

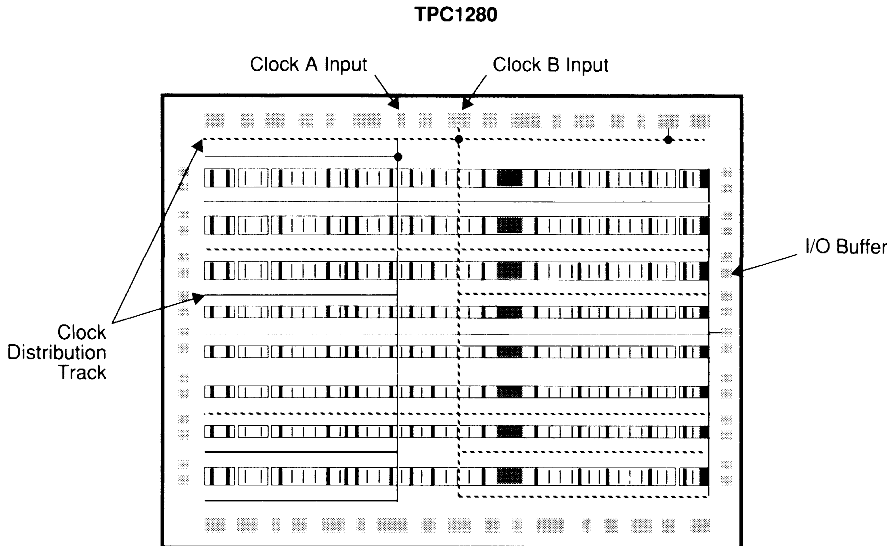


1.2.4.2 Clock Distribution

Clock distribution capabilities have also been enhanced for the TPC12 family. As Figure 1-17 shows, there are two separate clock distribution networks. Also, signals can get on the clock network either externally or

internally. This means that a clock signal can be conditioned before being fed to sequential modules on the device. This conditioning could include frequency division or duty cycle modification. Clock skew can be reduced even more on the TPC12 series part because the clock signal is driven from the center of the array. Typical skew is approximately 1–2 ns.

Figure 1-17. TPC12 Series Clock Distribution



1.2.4.3 Routing

Additional routing resources have also been added to TPC12 series parts. Thirty-six horizontal routing tracks are available, and vertical routing resources have been increased to 15 tracks.

1.2.5 Generating Logic Functions from Multiplexers

An interesting analysis to perform is the method of implementing various logic functions using a multiplexer-based architecture. This exercise demonstrates the flexibility of this scheme but is not a necessary function for you to perform.

The signal names used in this analysis are the same as those in Figure 1-18. It may help to refer to this figure as you read this section. The first step is to realize that the logic module is a 4-input multiplexer. Boolean simplification shows this to be true.

First, tie S0 and S1 together and call the signal B:

S0 + S1 = B + B = B

B is now the high address bit for the 4-input multiplexer. Then tie SA and SB together so that they become the low address bit and call this signal A. Now the equation for the multiplexer can be written as:

Y = A0/A/B + A1A/B + B0/AB + B1AB,

which is the equation for a 4-input multiplexer with signals A0, A1, B0, and B1 and address lines A and B.

Suppose the function implemented is:

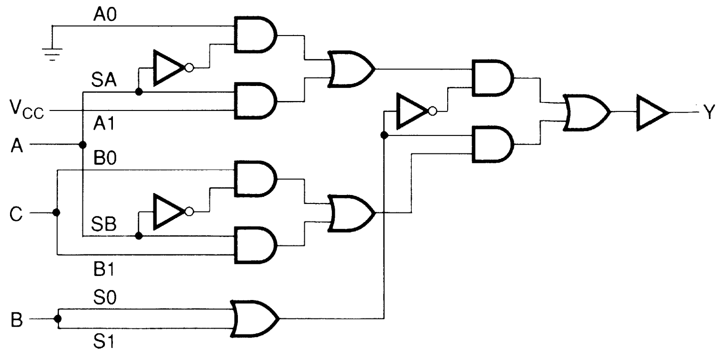
Y = /ABC + ABC + A/B.

This function can be expressed in Karnaugh Map form as:

	AB	00	01	11	10
C	0	0	0	0	1
	1	0	1	1	1

Each column in the map now corresponds to one AND function in the multiplexer with the AB address lines selecting the AND gate. The AND gate addressed by AB = 00 has output 0 regardless of C, so its input is tied to GND. The AND gates addressed by AB = 01 and 11 has an output that is the same as C so its input is tied to C. The last AND gate with address AB = 0 is always 1 so its input is tied to V_{CC}. This completes the implementation of the function. The completed circuit and logic module implementation is shown in Figure 1-18.

Figure 1-18. Logic Function Implementation



The TPC10 and TPC12 architectures provide a variety of resources which make them well suited to logic consolidation applications. The TPC10 family is a good choice when a single high-drive clock network is needed or highly efficient mapping of logic functions is required.

The TPC12 family is useful when two clock networks are needed, internal access to the clock network is required, many flip-flops are used, or high speed heterogeneous combinational-sequential circuits are implemented. In addition, the TPC12 family meets all the needs that the TPC10 family does plus offers higher density.

Chapter 2

FPGA Design Flow

This chapter discusses the FPGA design flow for various software utilities and supported hardware platforms.

2.1 Viewlogic on PC†

2.1.1 Introduction

Section 2.1 describes the features of the Viewlogic CAE system on a PC-386DX platform. This CAE system can be used to implement a design in a TI FPGA. A brief introduction to the design flow is given here, followed by a more detailed description based on a 5-bit synchronous counter application example. The simulator used is driven from its command language by a file. A comparison of both tabular and graphical output is given.

The following utilities from the Viewlogic's Workview™ suite of software are described: Viewfile™, Viewdraw™, Viewtext™, Viewsim™, and Viewwave™.

2.1.2 Viewlogic Design Flow

The basic Viewlogic design flow is illustrated in Figure 2-1. Elements of this flow are described as:

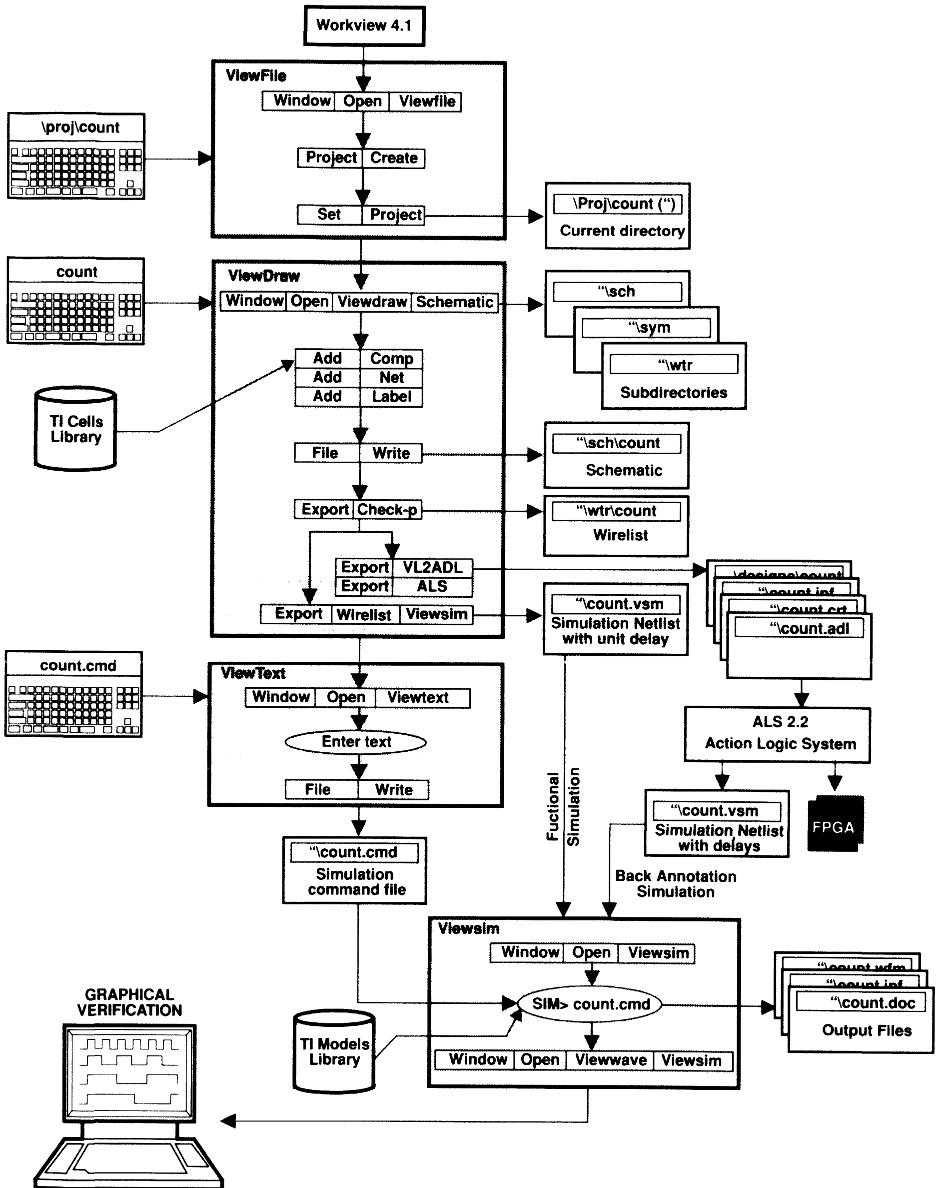
- Viewfile—File and project management
- Viewdraw—Schematic editor
- Viewtext—Text file editor
- Viewsim—Interactive logic simulator
- Viewwave—Interactive digital waveform processor

First, a project directory is set using Viewfile. This helps users to organize and maintain Workview's data files on a project basis. Having created the project directory, the Viewdraw schematic editor is used to build the design from the TI library of standard logic macros or user-defined macros. The diagram in Figure 2-1 shows that three subdirectories are created under the project for storing the `.sch`, `.sym`, and `.wir` files. Components, nets, and labels are added which build up the schematic. For large designs hierarchy can be used to make the design more manageable. Once the schematic is finished and checked for syntax errors, a simulation netlist is generated ready for simulation using Viewsim.

Before the simulator is used, a command file is generated which defines the format of the simulation and the stimulus to be applied to the inputs. Graphical information can be described in a textual format within the file. This file could also be generated from a waveform. Having a simulation netlist and command file, Viewsim is used to verify the design by performing a logical functional simulation. Both graphical and text outputs are obtainable from Viewsim, the graphical output is viewed using Viewwave and the text output can be viewed using Viewtext, or any other ASCII editor.

† Contributed by Doug Mackay, Technical Marketing, Texas Instruments Ltd.

Figure 2-1. Design Flow



After a successful functional simulation, the design is ready to export into TI-ALS. This is performed by returning to DOS and selecting the VL2ADL utility which converts the Workview netlist into an Advanced Design Language (ADL) netlist.

TI-ALS is then invoked and the design is placed and routed automatically. Before programming, the postlayout delays are extracted and inserted into the simulation netlist in Viewsim. This netlist is simulated again and the results compared with the prelayout simulation. Upon successful postlayout simulation the TI FPGA can be programmed using the TI Activator™.

This method of pre- and postlayout simulation ensures that all the necessary steps have been taken to achieve a *right-first-time* design approach, thus reducing the number of design iterations and allowing the product earlier entry to the market.

2.1.3 Creating a Schematic

This section shows how the schematics of the 5-bit counter application example were generated using Viewdraw.

Figure 2-2 shows the counter example built up from five D-type flip-flops and combinational logic. No optimization techniques have been used, apart from inverting some inputs to some gates. Figure 2-3 shows the top level of the design where the counter symbol is connected to the input and output pads of the FPGA. A top-down or bottom-up design method can be used for hierarchical designs.

Figure 2-2. Counter Example

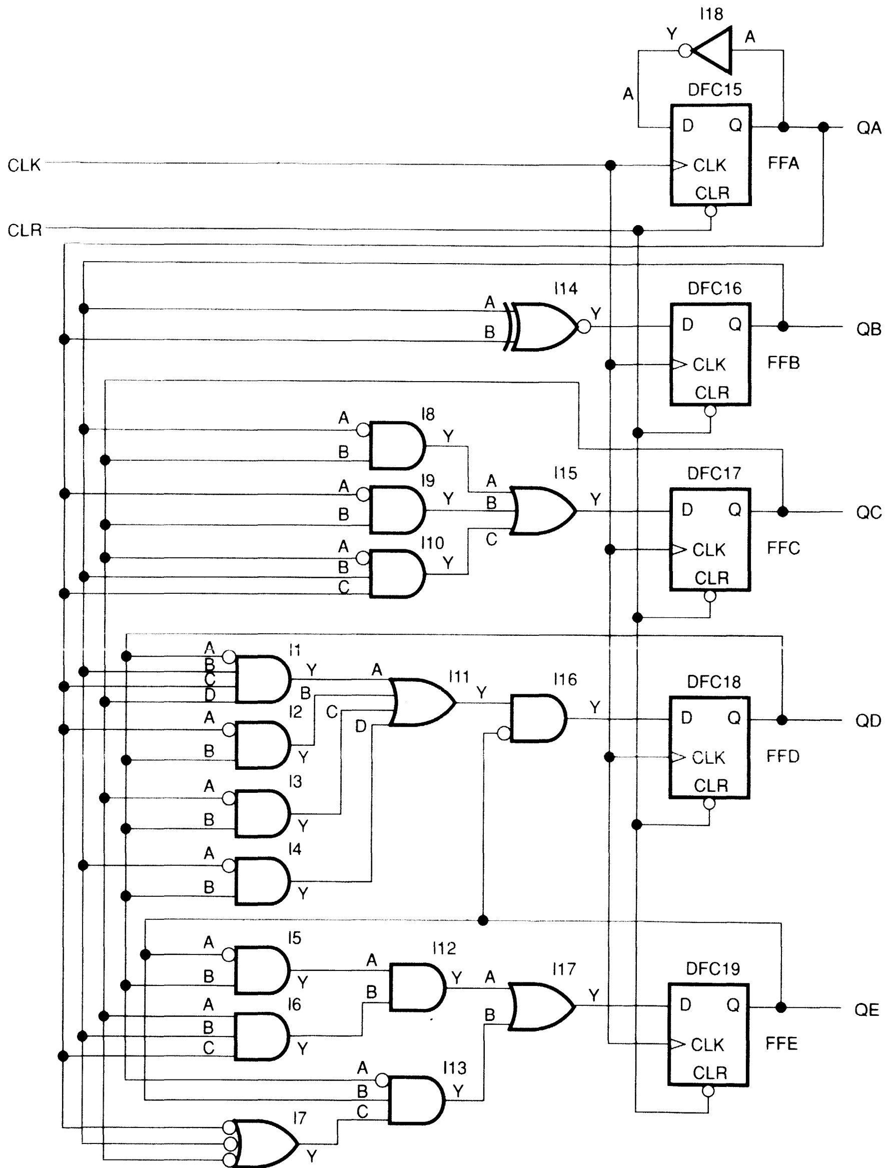
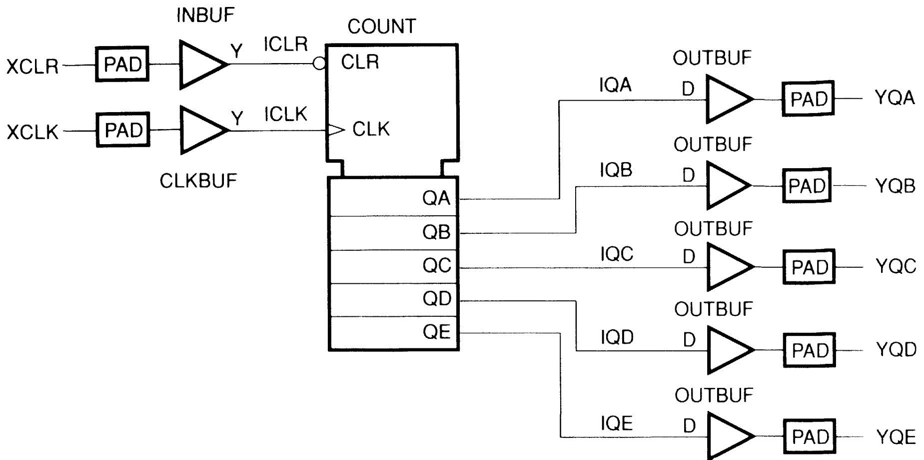


Figure 2-3. Top-Level Schematic



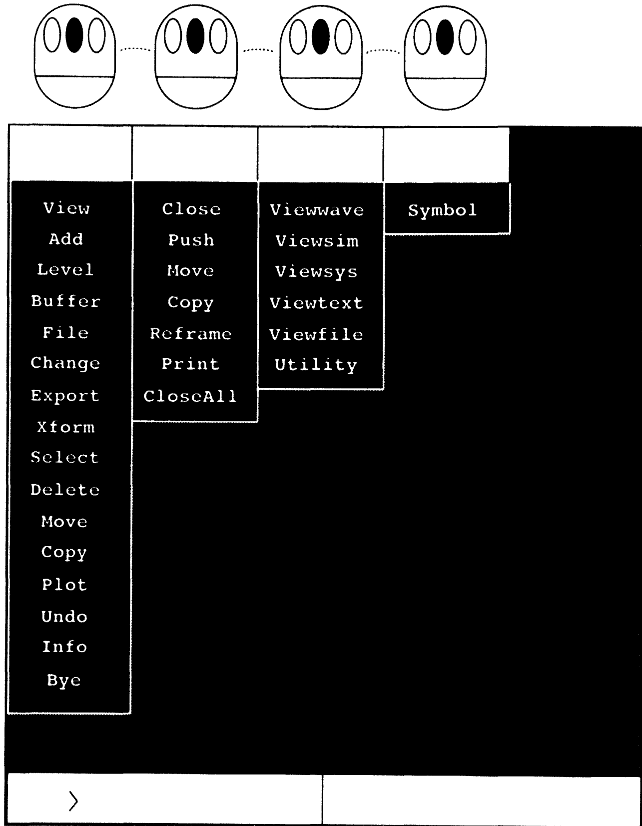
For the 5-bit counter shown in Figure 2-2 and Figure 2-3, a bottom-up design approach was used.

Before starting the design you need to set up the directory structure using Viewfile. For this example, `\proj\count` was used. Viewfile creates the necessary subdirectories and a `workview.ini` file. This `workview.ini` file is used to specify the initial setup and library search order for Workview. Viewfile can manage several projects but only one can be set as the working project.

Workview is a menu-based tool where each command is selected by the middle mouse button. Figure 2-4 shows the user-interface and mouse operation when opening a schematic.

Viewdraw is the Workview schematic editor. Figure 2-2 and Figure 2-3 show the example schematics which were drawn using Viewdraw.

Figure 2-4. Workview User Interface



After opening a new schematic sheet, the first part to the design process is to select the respective hard/soft macros from either the TPC10 series or TPC12 series of the TI FPGA library. For the schematic `count` there were 30 hard macros used. The menu option to pick a library component is `[Add|Component]`. If multiple hard macros are used then it's quicker to place one and make additional copies than to select each component from the library.

After placing the component instances, nets need to be added for continuity. The menu option `[Add|Net]` will allow nets to be drawn connecting schematic pins. The final stage in completing a schematic is to label all nets and components. This needs to be done to identify parts of the schematic

easily. Omitting labels will cause Workview to attach a meaningless five-character label to the component or net.

Saving the schematic using [File|Write] will perform a simple check and create a wirelist description of the schematic.

When the schematic sheet is completed, a symbol is generated allowing the sheet to be referenced higher in the hierarchy. For the example count the menu option [Window|Open|Viewdraw|Symbol] was used with the same name as the schematic. Once opened, a symbol body and pins were drawn and each pin labelled with the same name used in the schematic.

To call up this schematic/symbol [Add|Component] is selected from the menu and the symbol name given. Figure 2-3 shows the top level of the count design showing the user-created symbol count connected to the input and output pads. After the design is finished and checked, a schematic wirelist is generated from the menu option [Export|Wirelist|Viewsim]. This generates a count.vsm file ready for simulation.

2.1.4 Simulating a Schematic

Having generated a wirelist for simulation, Viewsim is opened in a similar way to the schematic using [Window|Open|Viewsim]. Simulation commands are executed from a command file. The command file COUNT.CMD for this design is shown in Figure 2-5. The .CMD file must be written by the user. Any PC ASCII editor can be used.

Figure 2-5. Example Simulation Command File

```
|=====================================================

|COUNT.CMD Simulation Command File for Viewsim for project
count

|===========================================================

|

Restart

|

| Set the step size to be 50nS and a '0 - 1' 1 MHz clock

stepsize 50ns

clock xclk 0 1
```

```
|
| Clear all registers with clr active for 50nS
wfm xclr @0ns=0 @50ns=1
|
| Setup vectors ALL and QBUS
vector qbus yqe yqd yqc yqb yqa
vector all xclk xclr yqa yqb yqc yqd yqe qbus
radix dec qbus all
|
| Every 90ns print values to file TRACE.OUT
| Generate a print on change table to TRACE.POC
watch xclk xclr yqa yqb yqc yqd yqe qbus
after 90ns do(every 100ns do(print > trace.out))
break all? do (print > trace.poc)
defaults -watch -time
|
| Generate a waveform file COUNT.WFM for viewwave
wave count.wfm xclk xclr yqa yqb yqc yqd yqe qbus
|
| Run 26 stepsizes of simulation
cycle 26
```

Executing the command file will simulate the design using the library models for the TI FPGA. Three files are generated from this command file, a `count.wfm` file is written, which is used by Viewwave for observing and verifying the results, and two trace files, generated to show simulation vectors.

The `trace.poc` file shows a print-on-change (POC) type of vector file where a vector is printed after a change in any signal. The `trace.out` file prints a vector after 90 ns into the 100 ns period. These are very useful for verifying large simulations and observing changes between pre- and

postlayout simulation. They can also be written into a suitable format to allow easy import into a logic tester as test vectors.

Figure 2-6 shows the listings of the prelayout and postlayout delays side by side. The effects of back annotating postlayout delays can be clearly seen in the results of the POC files.

Figure 2-6. Pre- and Postlayout `trace.poc` Vector Files

```
TRACE.POC Pre-Layout                 TRACE.POC  Post-Layout

           XXYYYYY Q                            XXYYYYY Q
           CCQQQQQ B                            CCQQQQQ B
           LLABCDE U                            LLABCDE U
           KR      S                            KR      S
         -----------                          -----------
      TIME BBBBBBB D                       TIME BBBBBBB D
     0.0ns 00XXXXX XX                     0.0ns 00XXXXX XX
     1.0ns 0000000 00                    29.3ns 000XXXX XX
    50.0ns 1100000 00                    31.7ns 0000XXX XX
    51.0ns 1110000 01                    32.1ns 0000X00 XX
   100.0ns 0110000 01                    32.5ns 0000000 00
   150.0ns 1110000 01                    50.0ns 1100000 00
   151.0ns 1101000 02                   100.0ns 0100000 00
   200.0ns 0101000 02                   150.0ns 1100000 00
   250.0ns 1101000 02                   168.5ns 1110000 01
   251.0ns 1111000 03                   200.0ns 0110000 01
   300.0ns 0111000 03                   250.0ns 1110000 01
   350.0ns 1111000 03                   268.6ns 1100000 00
   351.0ns 1100100 04                   270.8ns 1101000 02
   400.0ns 0100100 04                   300.0ns 0101000 02
   450.0ns 1100100 04                   350.0ns 1101000 02
   451.0ns 1110100 05                   368.5ns 1111000 03
```

```
 500.0ns 0110100 05
 550.0ns 1110100 05
 551.0ns 1101100 06
 600.0ns 0101100 06
 650.0ns 1101100 06
 651.0ns 1111100 07
 700.0ns 0111100 07
```

```
TRACE.OUT  Pre-Layout
         XXYYYYY Q
         CCQQQQQ B
         LLABCDE U
         KR      S
         -----------
    TIME BBBBBBB D
  90.0ns 1110000 01
 190.0ns 1101000 02
 290.0ns 1111000 03
 390.0ns 1100100 04
 490.0ns 1110100 05
 590.0ns 1101100 06
 690.0ns 1111100 07
 790.0ns 1100010 08
 890.0ns 1110010 09
 990.0ns 1101010 10
1090.0ns 1111010 11
1190.0ns 1100110 12
1290.0ns 1110110 13
1390.0ns 1101110 14
```

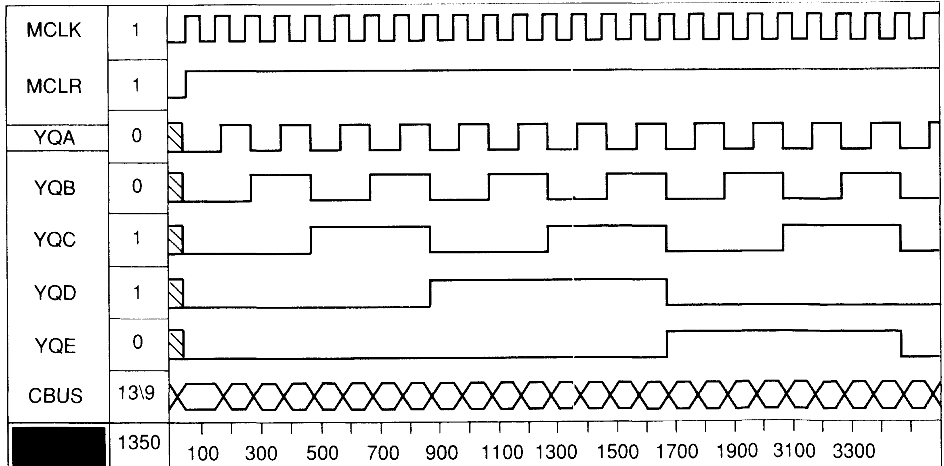
```
 400.0ns 0111000 03
 450.0ns 1111000 03
 468.6ns 1101000 02
 471.0ns 1100000 00
 472.2ns 1100100 04
 500.0ns 0100100 04
 550.0ns 1100100 04
```

```
TRACE.OUT  Post-Layout
         XXYYYYY Q
         CCQQQQQ B
         LLABCDE U
         KR      S
         -----------
    TIME BBBBBBB D
  90.0ns 1100000 00
 190.0ns 1110000 01
 290.0ns 1101000 02
 390.0ns 1111000 03
 490.0ns 1100100 04
 590.0ns 1110100 05
 690.0ns 1101100 06
 790.0ns 1111100 07
 890.0ns 1100010 08
 990.0ns 1110010 09
1090.0ns 1101010 10
1190.0ns 1111010 11
1290.0ns 1100110 12
1390.0ns 1110110 13
```

```
1490.0ns 1111110 15        1490.0ns 1101110 14

1590.0ns 1100001 16        1590.0ns 1111110 15

1690.0ns 1110001 17        1690.0ns 1100001 16

1790.0ns 1101001 18        1790.0ns 1110001 17

1890.0ns 1111001 19        1890.0ns 1101001 18

1990.0ns 1100101 20        1990.0ns 1111001 19

2090.0ns 1110101 21        2090.0ns 1100101 20

2190.0ns 1101101 22        2190.0ns 1110101 21

2290.0ns 1111101 23        2290.0ns 1101101 22

2390.0ns 1100000 00        2390.0ns 1111101 23

2490.0ns 1110000 01        2490.0ns 1100000 00
```

The waveform output `count.wfm` is viewed using the Viewwave utility. Figure 2-7 shows the graphical output of this. On the left are the plotted signal names, followed by the graphical waveform of this signal. A cursor is displayed on the screen and moved by the mouse. This cursor is used to measure timing information and display the wave values in the values' column on the display.

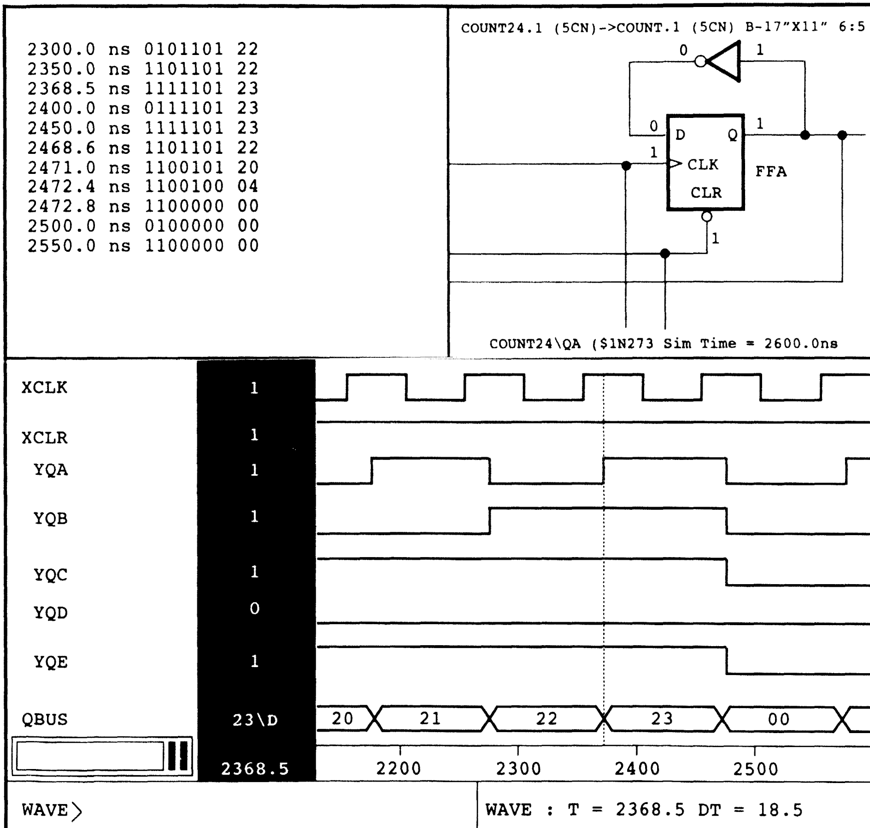
Figure 2-7. Simulation Waveform



This simulation was performed using the back annotated delay information which was extracted after place and route by the TI-ALS development system.

Multiple windows can be opened on the screen showing the results of the simulation and the schematic capture. An example of this is shown in Figure 2-8. In the schematic, values from the simulation are annotated into the schematic as logic values such as 1 or 0. This is a very useful feature for debugging designs, and the values of all hierarchical levels of the schematic are annotated. Figure 2-8 also shows the vector simulation output along with the graphical simulation output.

Figure 2-8. Workview Screen Layout



2.1.5 Generating an ADL Netlist

Generating the ADL netlist can be done by executing the VL2ADL utility from a DOS prompt.

Executing VL2ADL will generate three files.

`/designs/count/count.adl` (Exported ADL netlist)

`/designs/count/count.crt` (Blank critical path file)

`/designs/count/count.ipf` (Initial pin assignment file)

TI-ALS then needs to be used interactively to perform validation, I/O placement, place and route, and delay extraction. Postlayout simulation can be performed by back annotating the extracted delays into the Workview simulation netlist.

2.1.6 Postlayout Simulation

From within the TI-ALS, a delay file was automatically written containing all the net and component delays. This file is used to add delays into the simulation wirelist using the Workview VSM program.

With the new simulation wirelist, Viewsim is used in the same way as with unit delays. A modified command file can be used generating output files with additional delays added from layout. The Viewsim simulator is a very powerful tool which allows the user to verify the design before the need to program and lay out the design in silicon. After postlayout simulation is verified, the fuse file can then be used to program a TI FPGA using a TI Activator™ 1 or 2.

Section 2.1 has shown the PC Viewlogic design flow from entering a schematic to simulating with postlayout delays. A 5-bit counter example was used to show how a typical design would be implemented.

The powerful Viewsim simulator shows how easy it can be to verify large designs, thus reducing the need for testing a device in the application/test-rig. With postlayout simulation of the worst and best-case delay factors, you can simulate the design under realistic conditions to observe the effects on the design. If changes are necessary, they can be quickly made so as to save valuable time in the development of a product or system.

The suite of Workview software from Viewlogic is one of the most advanced PC-based FPGA CAD packages and is simple and efficient to use.

2.2 OrCAD on PC†

2.2.1 Introduction

The OrCAD™ CAE system is a reasonably priced and easy to use software package which supports FPGA design on a PC platform. The OrCAD system provides schematic capture, logic simulation, netlist generation, and rule checking, all of which are used in conjunction with the TI-ALS tools. The TI-ALS tools complete the functionality by providing rule checking, pin and module placement, net routing, static timing analysis, programming, and test. This section explains the design flow and system setup using this configuration for FPGA design.

2.2.2 PC Requirements

The TI-ALS version 2.2 software supports both OrCAD version 3.22 and OrCAD version 4.1. This section addresses the flow for version 4.1. Additional information on version 3.22 can be found in the *TI-ALS User's Guide*. TI provides OrCAD libraries as part of the TI-ALS system release. The OrCAD software must be obtained separately. Special hardware requirements include:

- 8 Mbytes of extended memory
- PC-386 or PC-486
- Sufficient hard disk space—which is usually about 25 Mbytes

2.2.3 `Autoexec.bat` and `config.sys`

The `autoexec.bat` file requires several entries to accommodate the OrCAD/TI-ALS flow. These entries are:

```
SET ALSDIR=E:ALS

SET HOME=E:ALSUSER

SET USRDIR=E:DESIGNS

SET USER=JDOE (or other appropriate name)

SET ORCADEXE=E:ORCADEXE

SET ORCADESP=E:ORCADESP

SET ORCADPROJ=E:ORCAD

SET ORCADUSER=E:ORCADESP
```

† Contributed by Joel S. Lason, P.E., FPGA Applications, Texas Instruments Incorporated.

```
PATH=E:ALSBIN;E:ORCADEXE;
```

Substitute the appropriate hard drive designator for your system (such as C:, D:, etc.).

The `config.sys` file requires the following entries:

```
files=30
```

```
buffers=30
```

`shell=c:command.com /p /e:1000` (or higher)

In addition, it may be necessary to type the line

`device=c:doshi+mem.sys`

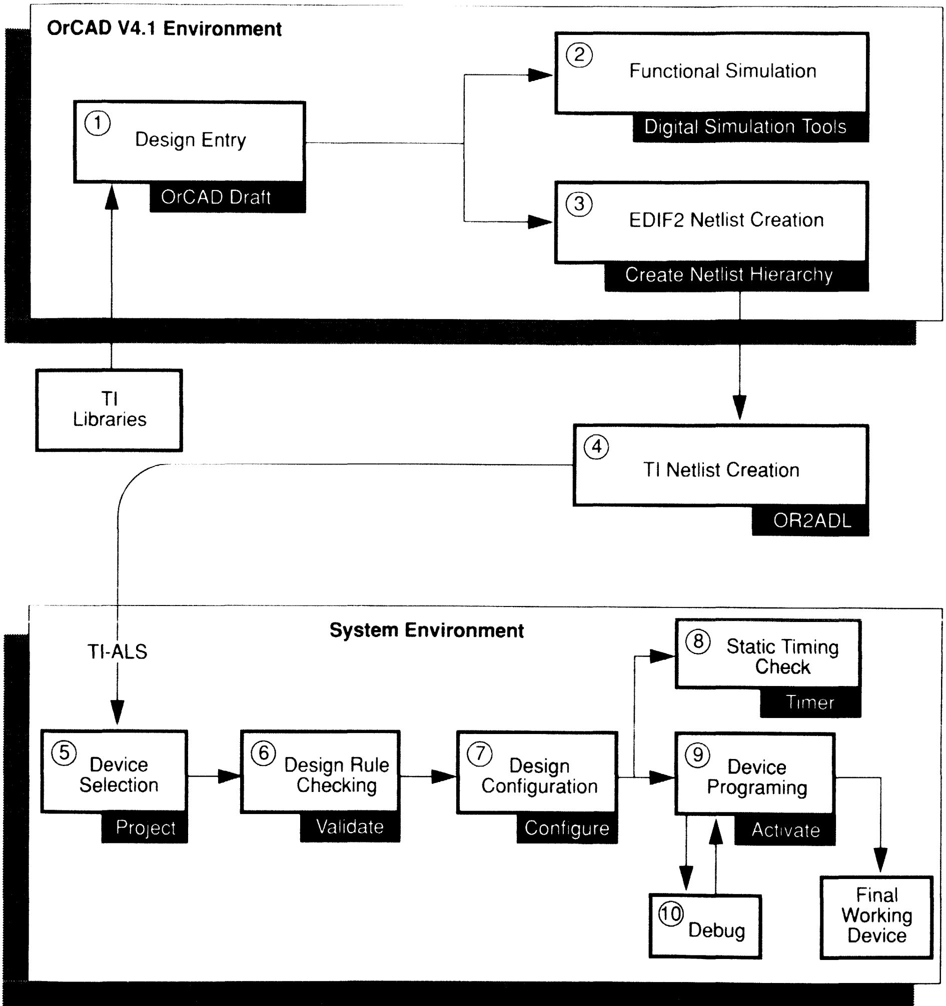
to avoid getting heap allocation errors from within the OrCAD environment. A specific place this can occur is in invoking the local configuration function for the Draft program, which will be discussed later.

It is necessary to reboot your system after making changes to the `autoexec.bat` and `config.sys` files for the changes to become effective.

2.2.4 OrCAD Design Flow

The TI FPGA design flow for OrCAD-based designs is shown in Figure 2-9. As the figure indicates there are two major divisions to the design flow: the OrCAD environment and the TI-ALS environment. The glue that holds these two environments together is the OR2ADL netlist translation program. OR2ADL is provided as part of the TI-ALS product release, which also includes the OrCAD schematic capture symbol and logic simulation model libraries.

Figure 2-9. TI-ALS/OrCAD Design Flow



2.2.5 Invoking OrCAD

OrCAD is invoked by typing `orcad` at the DOS prompt. This will bring up the ESP design environment, a graphical interface that controls the configuration of the various tools in the toolset and their execution.

2.2.6 Establishing a New Design

When starting a new design, the first step within ESP is creating the new design directory and configuration files. This is done by selecting the `Design Management Tools` menu option by clicking the left mouse button on it from within ESP. After invoking the design management tools, another menu will appear which has an entry called `Execute`. With the left mouse button select this entry to enter the design management tools.

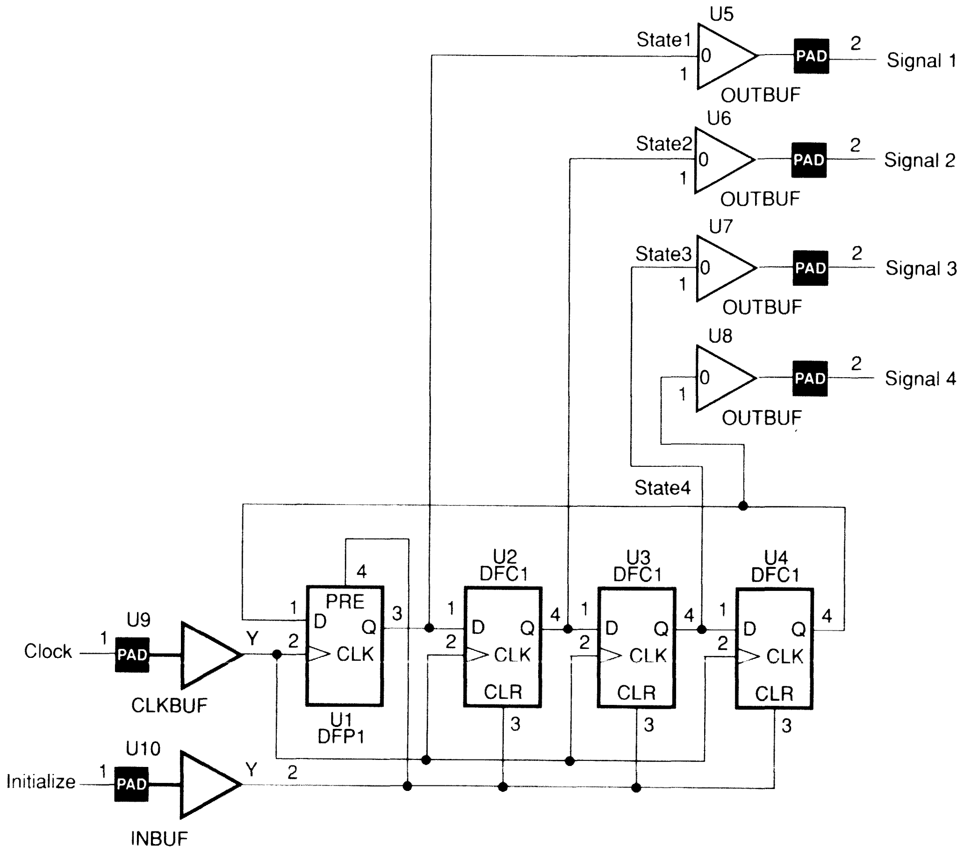
Another screen now appears which has the design management tools on it. At this point, select the `Create Design` option, provide the name of the design that you want to create, and click on the `OK` button. If no other actions are required, then select the `OK` button to exit design management tools. This makes the new design the active design.

If you enter OrCAD and the desired design is not active (the active design name appears in the banner at the top of the screen) but has been created, select `Design Management Tools`, and select `Execute`. Highlight the desired design by clicking on it with the left mouse button, and select `OK` to return to the main ESP screen.

2.2.7 Design Entry and Schematic Capture

Design entry is accomplished with the OrCAD Draft program. The Draft program is invoked by selecting `Schematic Design Tools` option from the main ESP screen. This will bring up another screen which lists the schematic design tools. In the upper left corner of the screen is the button (labeled `draft`) that you click to execute the Draft program. Selecting this button will invoke the schematic capture package using the root schematic file of the design. The name of the root schematic file will be the design name specified under the design management tools plus a `.sch` extension. A TPC10 series example design called `RINGCNTR` is used to illustrate the flow. Its schematic diagram is shown in Figure 2-10, and its name is `ringcntr.sch`.

Figure 2-10. OrCAD Schematic Diagram



Before schematic capture can proceed, several items need to be configured within the OrCAD toolset. As mentioned before, the ESP design environment allows this to be performed through various menu selections and on-screen forms. The configurations are summarized as follows:

- Under `Schematic Design Tools`,
  - Under `draft`, configure schematic tools as follows:

    Library options:

    library prefix: `als\lib1000*.lib`

Configured libraries:

flops.lib

flops2.lib

gates1.lib

gates2.lib

gates3.lib

power.lib

soft.lib

- Under Annotate schematic, setup the local configuration to configure annotate with the source file as the root of the design.
- Under Create Hierarchical Netlist | Local Configuration, do the following:

  Configure INET processing options

  - Descend into sheetpath parts
  - Unconditionally process all sheets in design

  Configure HFORM processing options

  - Format prefix/wildcard: alsdata*.ch
  - Selected format: alsedif.ch

This configuration varies only slightly for TPC12 series designs and the differences are documented in the TI-ALS FPGA user documentation.

To begin schematic capture, select the **draft** command from the Schematic Design Tools screen. This will bring up the root page of the design. Several highly useful command sequences that do most of the work for a single page design are:

❑ Instantiating a component

Click the left mouse button to bring up the main command menu from which the **get** command is invoked. Move the highlight bar down to the **get** command and click the left mouse button again. You will then be prompted for a component name with the Get? prompt. Enter the component name or click the left mouse button again to bring up a list of the configured libraries. Scroll through the list again with the mouse and select the appropriate library. A new menu will now appear with a list of component names. Select a part, move the part to the desired location and execute the **place** command with the mouse.

The sequence indicated is:

[Get] Get? (Click Left) [flop.lib][dfl] (Move Mouse) [Place]

This will place a DF1 macro from the `flop.lib` library on the page. Additional command sequences will be specified in this format.

- Assigning reference designators.

  `[Edit]` (Move to Component) `[Edit] [Reference] [Name] Name?`

- Entering wires.

  `[Place] [Wire] [Begin] [End]`

- Showing connections between wire segments.

  `[Place] [Junction]`

Note:

This is used to show electrical connection when wires meet at a cross-section or junction.

- Adding labels to wires.

  `[Place] [Label] Label?`

Note:

The lower left corner of the first letter is the attachment point and must be located on the wire to be labelled.

- Adding module ports.

  `[Place] [Module Port] Module Port Name? [Input or Output or Bidir] [Place]`

Note:

The module port name will become the name of the pin.

- Exiting and saving.

  `[Quit] [Update File] [Abandon Edits]`

2.2.8 Logic Simulation

Logic simulation is accomplished with the OrCAD Simulate program. To invoke the Simulate program, select the `Digital Simulation Tools` button with the mouse and then select the `Execute` option from the pull-down menu. The `Digital Simulation Tools` screen now appears.

There is a box labeled `editors` on this screen and it has a button within it labeled `simulate`. Select this button to invoke the simulator. If, however, this is the first time to run simulation for this design, then the digital simulation tools will have to be configured first. The main item that will have to be configured is the device model library. The required library for FPGA simulation is provided with the TI-ALS release. It can be found in the directory:

```
e:\als\lib\a1000
```

and the correct filename is `act1.dsf`. Copy this file to the directory:

```
e:\orcadesp\vst\library
```

under the same name. In the `Digital Simulation Tools` screen, under the box labeled `librarians`, there is a button for the `Add Device Model` option. Select this button. Then select the menu entry labelled `Local Configuration`. Finally, accept the `Configure Modelpro` menu selection. For a TPC10 series device, type

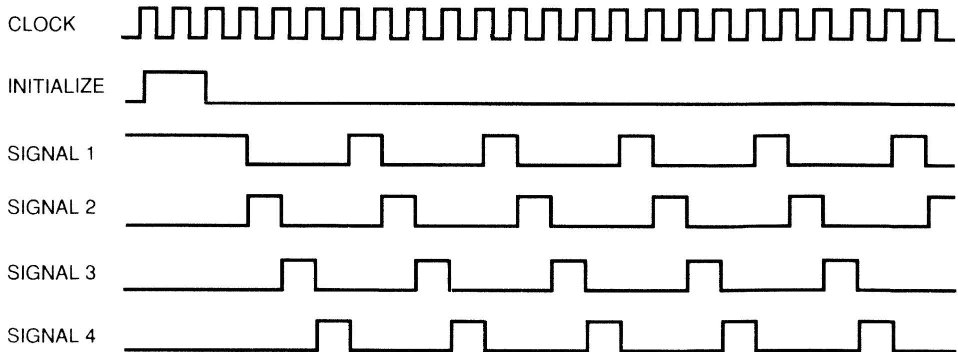
`e:orcadespvst\library*.dsf`

in the box labeled `Prefix/Wildcard`. Then select `act1.dsf` from the listing of libraries. Select `OK` from the top of the local configuration box to exit from the local configuration screen.

So far, only the configuration of `Add Device` has been performed. `Add Device` itself still needs to be executed. Simply click the mouse pointer on `Add Device` to make the TI FPGA libraries active. To configure the libraries for TPC12 series designs, consult the TI FPGA and TI-ALS user documentation. It is a similar process but with different filenames.

The digital simulation tools are now configured for TI FPGAs. To enter the simulator select the `simulate` button and then select the `execute` option that appears in the menu. As the simulator loads it will indicate that it needs stimulus and trace files. These functions can be performed within this context and are detailed in the *OrCAD Digital Simulation Tools User's Guide*. The remaining functions to be performed are also well documented in the OrCAD and TI FPGA user documentation and so are not covered here. The sample simulation output for the `RINGCNTR` design is shown in Figure 2-11.

Figure 2-11. Simulation Waveforms



2.2.9 Netlist Translation

After the design has been captured and the functionality has been verified with the digital simulation tools, it is ready for translation to a TI-ALS compatible netlist (ADL format). This consists of several steps:

1) Electrical rules checking

2) Schematic cleanup

3) OrCAD to EDIF 2.0.0 translation

4) EDIF 2.0.0 to ADL netlist translation

Electrical rules checking is performed within the ESP design environment under the `Schematic Design Tools` menu. A button is selected which performs this check automatically. It is possible to configure the types of checks performed within the configure schematic tools menu. The default values are generally acceptable. Electrical rules checking evaluates problems such as open input pins, shorts, and bus contention. The result is a file which contains both errors and warnings. Errors must be corrected but warnings might not need correction.

Schematic cleanup is also performed from within the schematic design tools menu of the ESP environment. This function essentially checks for drafting errors such as overlapping and duplicate objects.

The EDIF 2.0.0 netlist is created by invoking the `Create Hierarchical Netlist` function from within the schematic design tools menu. The configuration items specified previously have taken care of the setup requirements for this function. All that remains is to push the button and let

the computer generate the output. This output file will be written to the design directory created when the design management tool was executed and will have a .net extension. For this design the directory will be e:orcadringcntr. The EDIF netlist for this design is provided in Figure 2-12.

It is now time to exit OrCAD and perform the translation from EDIF 2.0.0 to a TI-ALS compatible .adl netlist. From the schematic design tools menu select the To Main function and then select the Exit ESP function.

Note:

It may be necessary at this time to reconfigure the system for TI-ALS. One specific case involves memory managers. If the MS-DOS™ himem.sys program was loaded for OrCAD it must be removed before generation of the ADL netlist and the invocation of the TI-ALS tool set. TI-ALS provides its own memory management capability.

After exiting OrCAD, change to the OrCAD project directory, which, as mentioned earlier, is the e:orcadringcntr directory. Do not confuse the OrCAD project directory with the TI-ALS designs directory. OR2ADL is the program which performs the translation from EDIF 2.0.0 to ADL format. The syntax for invoking the translation is:

```
OR2ADL [fam:act1] design_name
```

ACT1 is used for TPC10 series designs while ACT2 is used for TPC12 series designs. A sample ADL netlist for this design can be found in Figure 2-13. After successful translation, invoke TI-ALS and process the design through the standard flow as indicated in the TI Action Logic System Overview found in this handbook. If insufficient memory errors occur during ADL netlist creation then you may have a problem with your config.sys file. Specifically, the himem.sys driver may still be installed.

Figure 2-12. EDIF 2.0.0 Netlist for the `RINGCNTR` *Design*

```
(edif RINGCNTR

(edifVersion 2 0 0)
  (edifLevel 0)
  (keywordMap (keywordLevel 0))
 (status
  (written
  (timeStamp 0 0 0 0 0 0)
    (program "HFORM.EXE")
  (comment "Original data from OrCAD/SDT schematic"))
   (comment "")
   (comment "September  2, 1992")
   (comment "")
   (comment "")
   (comment "")
   (comment "")
   (comment "")
   (comment "")
   (comment ""))
  (external OrCAD_LIB
   (edifLevel 0)
   (technology
    (numberDefinition
     (scale 1 1 (unit distance))))
   (cell CLKBUF
    (cellType generic)
    (comment "From OrCAD library GATES1.LIB")
    (view NetlistView
     (viewType netlist)
     (interface
      (port PAD (direction INPUT))
      (port Y (direction OUTPUT)))))
   (cell DFC1
    (cellType generic)
    (comment "From OrCAD library FLOPS.LIB")
    (view NetlistView
     (viewType netlist)
     (interface
      (port D (direction INPUT))
      (port Q (direction OUTPUT))
      (port CLK (direction INPUT))
      (port CLR (direction INPUT)))))
   (cell DFP1
    (cellType generic)
    (comment "From OrCAD library FLOPS.LIB")
    (view NetlistView
     (viewType netlist)
     (interface
```

```
    (port D (direction INPUT))
    (port Q (direction OUTPUT))
    (port CLK (direction INPUT))
    (port PRE (direction INPUT)))))
 (cell INBUF
  (cellType generic)
  (comment "From OrCAD library GATES1.LIB")
  (view NetlistView
   (viewType netlist)
   (interface
    (port PAD (direction INPUT))
    (port Y (direction OUTPUT)))))
 (cell OUTBUF
  (cellType generic)
  (comment "From OrCAD library GATES3.LIB")
  (view NetlistView
   (viewType netlist)
   (interface
    (port D (direction INPUT))
    (port PAD (direction OUTPUT))))))
(library MAIN_LIB
 (edifLevel 0)
 (technology
  (numberDefinition
   (scale 1 1 (unit distance))))
 (cell RINGCNTR
  (cellType generic)
  (view NetlistView
   (viewType netlist)
   (interface
    (port CLOCK (direction INPUT))
    (port INITIALIZE (direction INPUT))
    (port SIGNAL1 (direction OUTPUT))
    (port SIGNAL2 (direction OUTPUT))
    (port SIGNAL3 (direction OUTPUT))
    (port SIGNAL4 (direction OUTPUT)))
   (contents
    (instance U1
     (viewRef NetlistView
      (cellRef DFP1
(libraryRef OrCAD_LIB)))
     (property PartValue (string "DFP1"))
     (property ModuleValue (string "DFP1")))
    (instance U2
     (viewRef NetlistView
      (cellRef DFC1
(libraryRef OrCAD_LIB)))
     (property PartValue (string "DFC1"))
     (property ModuleValue (string "DFC1")))
    (instance U3
```

```
    (viewRef NetlistView
     (cellRef DFC1
(libraryRef OrCAD_LIB)))
    (property PartValue (string "DFC1"))
    (property ModuleValue (string "DFC1")))
   (instance U4
    (viewRef NetlistView
     (cellRef DFC1
(libraryRef OrCAD_LIB)))
    (property PartValue (string "DFC1"))
    (property ModuleValue (string "DFC1")))
   (instance U5
    (viewRef NetlistView
     (cellRef OUTBUF
(libraryRef OrCAD_LIB)))
    (property PartValue (string "OUTBUF"))
    (property ModuleValue (string "OUTBUF")))
   (instance U6
    (viewRef NetlistView
     (cellRef OUTBUF
(libraryRef OrCAD_LIB)))
    (property PartValue (string "OUTBUF"))
    (property ModuleValue (string "OUTBUF")))
   (instance U7
    (viewRef NetlistView
     (cellRef OUTBUF
(libraryRef OrCAD_LIB)))
    (property PartValue (string "OUTBUF"))
    (property ModuleValue (string "OUTBUF")))
   (instance U8
    (viewRef NetlistView
     (cellRef OUTBUF
(libraryRef OrCAD_LIB)))
    (property PartValue (string "OUTBUF"))
    (property ModuleValue (string "OUTBUF")))
   (instance U9
    (viewRef NetlistView
     (cellRef CLKBUF
(libraryRef OrCAD_LIB)))
    (property PartValue (string "CLKBUF"))
    (property ModuleValue (string "CLKBUF")))
   (instance U10
    (viewRef NetlistView
     (cellRef INBUF
(libraryRef OrCAD_LIB)))
    (property PartValue (string "INBUF"))
    (property ModuleValue (string "INBUF")))
   (net STATE1_1
    (joined
     (portRef Q (instanceRef U1))
```

```
  (portRef D (instanceRef U5))
  (portRef D (instanceRef U2))))
(net SIGNAL1
 (joined
  (portRef SIGNAL1)
  (portRef PAD (instanceRef U5))))
(net STATE2_1
 (joined
  (portRef Q (instanceRef U2))
  (portRef D (instanceRef U6))
  (portRef D (instanceRef U3))))
(net SIGNAL2
 (joined
  (portRef SIGNAL2)
  (portRef PAD (instanceRef U6))))
(net STATE3_1
 (joined
  (portRef Q (instanceRef U3))
  (portRef D (instanceRef U7))
  (portRef D (instanceRef U4))))
(net SIGNAL3
 (joined
  (portRef SIGNAL3)
  (portRef PAD (instanceRef U7))))
(net STATE4_1
 (joined
  (portRef D (instanceRef U1))
  (portRef D (instanceRef U8))
  (portRef Q (instanceRef U4))))
(net SIGNAL4
 (joined
  (portRef SIGNAL4)
  (portRef PAD (instanceRef U8))))
(net INIT_1
 (joined
  (portRef Y (instanceRef U10))
  (portRef PRE (instanceRef U1))
  (portRef CLR (instanceRef U2))
  (portRef CLR (instanceRef U3))
  (portRef CLR (instanceRef U4))))
(net CLOCK
 (joined
  (portRef CLOCK)
  (portRef PAD (instanceRef U9))))
(net OSC_1
 (joined
  (portRef CLK (instanceRef U2))
  (portRef Y (instanceRef U9))
  (portRef CLK (instanceRef U1))
  (portRef CLK (instanceRef U3))
```

```
        (portRef CLK (instanceRef U4))))
      (net INITIALIZE
       (joined
        (portRef INITIALIZE)
        (portRef PAD (instanceRef U10))))))))
  (design RINGCNTR
   (cellRef RINGCNTR
    (libraryRef MAIN_LIB))))
```

Figure 2-13. Sample ADL Netlist

```
; HEADER
; FILEIDADLe:\designs\ringcntr\ringcntr.adl 7cd4f9ad
; CHECKSUM 7cd4f9ad
; PROGRAM cae2adl
; VERSION 2.11
; NODEID 00005408
; VAR DEFSYS e: ls/data/system.def
; VAR DEFUSR e: lsuserjdoejdoe.def
; VAR DEFDES e:\designs raffic raffic.def
; VAR DEFDES e:\designs\ringcntr\ringcntr.def
; VAR QUIETDEFOPENFAIL <NOT-SET>
; VAR DESDIR e:\designs\ringcntr
; VAR DESIGN ringcntr
; VAR FUNC <NOT-SET>
; VAR MSGLENGTH <NOT-SET>
; VAR MANUFACTURER
; VAR XBATMODE <NOT-SET>
; VAR CAE or4
; VAR ALPHA_NAME <NOT-SET>
; VAR BINDIR e: ls\bin
; VAR noAlsPgCnst <NOT-SET>
; VAR OrEdifCfg e: ls\data lsedif.ch
; VAR QUIETDEFOPENFAIL 1
; VAR ALSVERBOSE <NOT-SET>
; VAR ALSDIR e: ls
; VAR MSGINDEX e: ls\data ls211.idx
; VAR MSGTEXT e: ls\data ls211.txt
; VAR USERHOME e: lsuserjdoe
; VAR FAM ACT1
; VAR ALLFAMS ACT1 ACT2
; VAR ACT1 1000
; VAR ADLIB e: ls\data 1000 dl05.lib
; VAR USRDIR e:\designs
; VAR DIE <NOT-SET>
; VAR PACKAGE <NOT-SET>
; VAR IPF e:\designs\ringcntr\ringcntr.ipf
; VAR CRT e:\designs\ringcntr\ringcntr.crt
; VAR CONVRTFLAG
; VAR OR_INET_ARGS <NOT-SET>
```

```
; VAR CAE2ADLNOVB <NOT-SET>
; VAR XCHKEXT <NOT-SET>
; VAR OR_HFORM_ARGS <NOT-SET>
; VAR AAL e:\designs\ringcntr\ringcntr.aal
; VAR ADL e:\designs\ringcntr\ringcntr.adl
; VAR DELETEINSTPROPS LEVEL
; VAR PERMITBADNAMES <NOT-SET>
; VAR FormLogAppend <NOT-SET>
; ENDHEADER
DEF RINGCNTR; SIGNAL4, SIGNAL3, SIGNAL2,

SIGNAL1, INITIALIZE,
   CLOCK.
USE ADLIB:INBUF; U10.
USE ADLIB:OUTBUF; U8.
USE ADLIB:OUTBUF; U7.
USE +IB:DFC1; U3.
USE ADLIB:DFC1; U2.
USE ADLIB:DFP1; U1.
USE ADLIB:CLKBUF; U9.
USE ADLIB:OUTBUF; U5.
USE ADLIB:OUTBUF; U6.
USE ADLIB:DFC1; U4.
NET INITIALIZE;  INITIALIZE, U10:PAD.
NET OSC_1;  U2:CLK, U9:Y, U1:CLK, U3:CLK,

U4:CLK.
NET CLOCK;  CLOCK, U9:PAD.
NET INIT_1;  U2:CLR, U10:Y, U1:PRE, U3:CLR,

U4:CLR.
NET SIGNAL4;  SIGNAL4, U8:PAD.
NET STATE4_1;  U1:D, U8:D, U4:Q.
NET SIGNAL3;  SIGNAL3, U7:PAD.
NET STATE3_1;  U7:D, U3:Q, U4:D.
NET SIGNAL2;  SIGNAL2, U6:PAD.
NET STATE2_1;  U6:D, U2:Q, U3:D.
NET SIGNAL1;  SIGNAL1, U5:PAD.
NET STATE1_1;  U5:D, U1:Q, U2:D.
END.
```

2.3 Valid on Sun[†]

2.3.1 Introduction

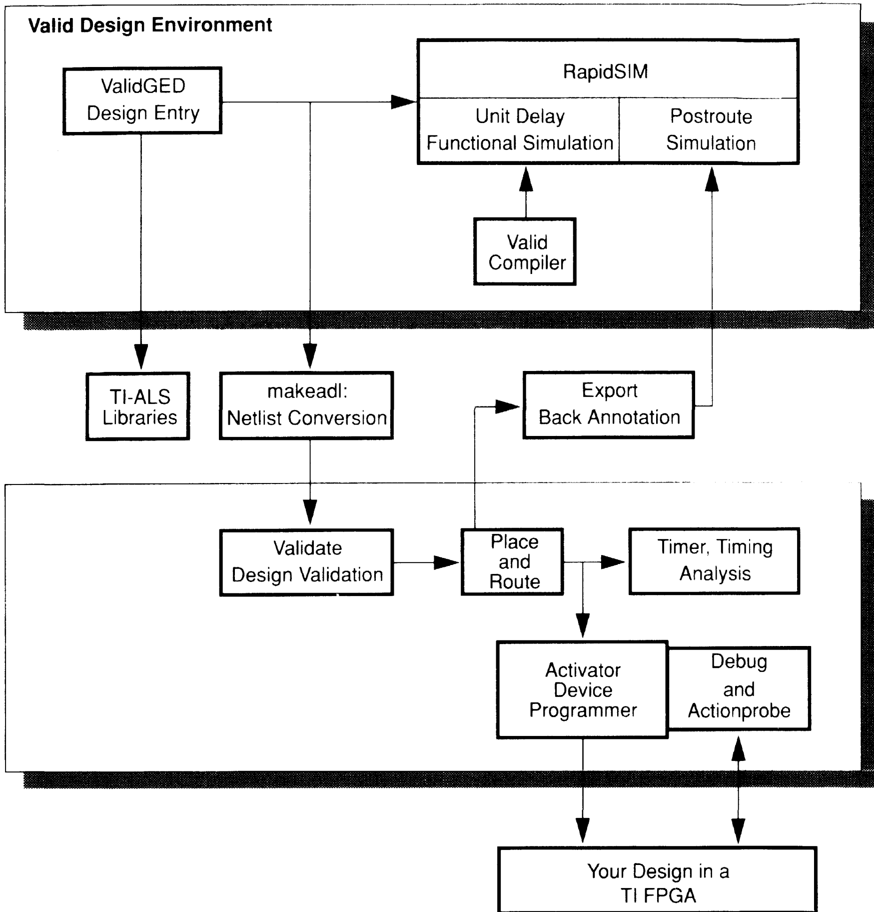
The purpose of Section 2.3 is to outline the TI-ALS design flow when using the Valid schematic capture and simulation software packages on a Sun™ workstation. Refer to the Valid tutorials and manuals to learn how to use the Valid software.

The general TI-ALS design flow using Valid is illustrated in Figure 2-14. ValidGED™ is used to enter the schematic. The schematic files may be used to create either a functional simulation netlist or a TI-ALS compatible netlist. The simulation netlist is created by using the ValidCOMPILER™. The TI-ALS compatible netlist is created by using the VA2ADL utility.

Next, the TI-ALS Validate program is used to check the design for design rule violations, and then the TI-ALS Configure program automatically places and routes the design. The TI-ALS Timer utility can then be used for static timing analysis, or a back annotation file may be created for Valid's RapidSIM™ simulator to run accurate system simulations. After satisfactory timing and simulation analyses, the TI FPGA is programmed using the Activator 1 or Activator 2 programming units.

[†] Contributed by FPGA Applications Staff, Texas Instruments Incorporated.

Figure 2-14. TI-ALS Valid Design Flow



Note:

The Activator 1 programming unit will interface only with a PC-386DX or PC-486DX and can program one unit at a time. The Activator 2 programming unit will interface with a PC-386DX or PC-486DX, Sun workstation, and HP/Apollo™ workstation and can program up to four units at a time.

2.3.2 Directory Structure and Symbolic Links

The TI-ALS software searches for executable and design files in the /usr/als directory. If you wish to install the TI-ALS software in a directory other than /usr/als you must create a symbolic link from /usr/als to the directory that contains the TI-ALS software.

For example, if the TI-ALS software is installed in the /home1/ti the following symbolic link must be created by your typing:

```
ln -s /home1/ti/als /usr/als
```

2.3.3 TI-ALS/Valid Library Files

Two TI-ALS libraries are used with Valid Logic Systems CAE design software. The libraries contain all the symbol (.body.1.n), schematic (.logic.1.n), and simulation (.sim.1.n) files necessary for schematic capture and simulation of a TI FPGA. The ACT1 library is used for TPC10 designs and the ACT2 library is used for TPC12 designs. The default location for these libraries is in the /usr/als/lib/va/act1 and /usr/als/lib/va/act2 directories.

The installation will create a symbolic link from /usr/valid/lib/va/act1 to /usr/als/lib/va/act1 and from /usr/valid/lib/va/act2 to /usr/als/lib/va/act2. You must add the following links to the /usr/valid/lib/master.lib file to reflect the current library structure:

```
'act1' '/usr/valid/lib/va/act1/act1.lib'

'act2' '/usr/valid/lib/va/act2/act2.lib'
```

All files for the design are stored in the same design directory. The Valid software files include the startup.ged, compiler.cmd, and simulate.cmd files, as well as the symbol (.body) and schematic (.logic) files for your design. The ValidCOMPILER produces the cmpexp.dat and cmpsyn.dat files that are used to create the TI-ALS compatible (.adl) netlist file.

2.3.4 ValidGED Schematic Editor

For each TI-ALS design, modify the startup.ged file so that either the A1000 or the A1200 library is specified for access by ValidGED. Designs cannot use TPC10 and TPC12 macros simultaneously, so only one of the libraries can be accessed in any particular startup.ged file.

Each new design must have a directory where all the TI-ALS and Valid files for the design will reside. Directory and schematic names must begin with a letter and be no longer than eight alphanumeric characters.

> The MAKEADL utility program will not function successfully if the underscore (_) character is used in the design name.

2.3.5 TI-ALS Conventions for Schematic Capture Using ValidGED

These TI-ALS design conventions apply when entering a schematic:

- The schematic must contain only components from the Valid standard library and the TI-ALS A1000 or A1200 libraries.
- Net names that are specified by the **`signame`** command must begin with a letter. In addition to alphanumeric characters, the following characters are allowed: @,#,%,^,&,_,-,+,=,~, and ?.
- You must label all bus signals that originate from merge bodies in the Valid standard library with user-defined names. Use the **`signame`** command in ValidGED to define the bus labels. You cannot leave buses from merges unlabeled, because the ValidCOMPILER will create default net names that are incompatible with the TI-ALS system software.
- You may indicate power and ground signals by attaching either the V_{CC} and GND symbols, or the 0 and 1 signal names to a wire.
- Only one output can drive a particular net in the design. Thus, wired-AND and wired-OR configurations are not allowed.
- The size and times properties are not allowed. This ValidGED feature allows a single schematic component to represent multiple components, much like a bus represents multiple nets. The TI-ALS macro library does not permit a single component to represent multiple components.
- Internal and external signals must pass through an I/O buffer macro prior to going on or off-chip. The I/O buffers must be located in the top-level schematic of the hierarchy.
- The EXOR and BUFF macros replace the XOR and BUF macros, respectively, in the Valid/TI-ALS library. The XOR and BUF macros are functionally identical to EXOR and BUF.

2.3.6 Adding Title and Abbreviation Properties

Valid and TI-ALS both require title and abbreviation properties. A maximum of eight characters and an abbreviation are added to each schematic. For each schematic in your design, add a drawing body with title and abbreviation properties attached.

2.3.7 Modifying TI-ALS Soft Macros

The TI-ALS macro libraries contain soft macros. These are higher-level logic functions created from standard hard macros. It is possible to copy the schematics and symbols of TI-ALS soft macros, modify them, and save them as user-defined macros. To do this, use the **write** or **diagram** commands to create a copy of the TI soft macro. Then, you can tailor the function of the new soft macro to your design's requirements. Remember to use the **change** command to modify the title and abbreviation properties of the drawing body so that it matches the new macro name.

2.3.8 Top-Level Symbols

A symbol (`.body`) drawing of the top-level schematic is not necessary for TI FPGA designs. The top-level symbols are only necessary when an FPGA is used in board-level schematics and simulations.

2.3.9 Unit-Delay Functional Simulation

After schematic capture, the design is ready for unit-delay simulation. Compile the design using the ValidCOMPILER, then enter the Valid simulator. The delay through each level of logic will be one nanosecond.

2.3.10 ADL Netlist Conversion

To create the TI-ALS compatible (`.adl` file) netlist and other TI-ALS design files, execute the **va2adl** ***<design name>*** command in the design directory. The command **va2adl** compiles the design using the ValidCOMPILER then converts the Valid netlist files `cmpexp.dat` and `cmpsyn.dat` into TI-ALS netlist (`.adl`), pin (`.ipf`), and criticality (`.crt`) files. The correct syntax for the **va2adl** command is:

va2adl [fam:*<family_name>*] *<design_name>*

where *<family_name>* is `act1` or `act2`.

2.3.11 Postroute Simulation with Back Annotated Delays

After place and route, verifying the FPGA ac performance requires extracting physical timing delay information from TI-ALS and back annotating wire delays to the Valid simulator. After the design has been placed and routed with the TI-ALS Configure programs, you can create a back annotation file using the **export** command in TI-ALS. This creates the `design_name.min` file and the `design_name.max` file. These files contain the back annotated timing information of the design for the Valid simulator. To use one of these

files in your simulation, edit the `simulate.cmd` file in your *<design name>* directory and add the lines:

`wire_delays` ***<design_name.min>*** or

`wire_delays` ***<design_name.max>***

The TI-ALS system used in conjunction with the Valid tool set provides a powerful desktop FPGA design solution. This fully integrated approach offers you the ability to utilize Valid's popular design entry and design verification tools coupled with the TI-ALS system's automatic device configuration and programming features to quickly turn your design concept into programmed silicon.

2.4 Viewlogic on Sun†

2.4.1 Introduction

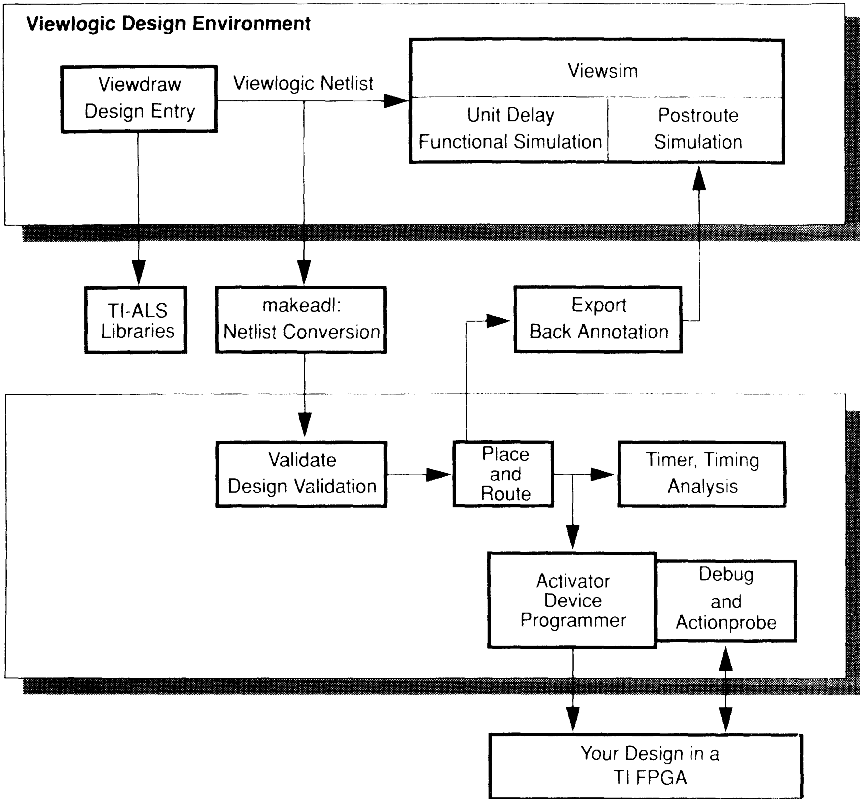
Section 2.4 outlines the TI Action Logic System (TI-ALS), the design flow during use of the Viewlogic schematic capture, and the simulation software packages that run on Sun workstations. Refer to the Viewlogic tutorials and manuals to learn how to use the Viewlogic software.

The general TI-ALS design flow using Viewlogic is illustrated in Figure 2-15. Viewdraw is used to enter the schematic. The schematic files may be used to create either a functional simulation netlist or a TI-ALS compatible netlist. The simulation netlist is created by selecting `Export|Wirelist|Viewsim` from the Viewlogic menus. The TI-ALS compatible netlist is created by using the **`vl2adl`** command.

Next, the TI-ALS Validate program is used to check the design for design rule violations, and then the TI-ALS Configure program automatically places and routes the design. The TI-ALS Timer utility can then be used for static timing analysis, or a back annotation file may be created for Viewlogic's Viewsim simulator to run accurate system simulations. After satisfactory timing and simulation analyses, the TI FPGA is programmed using a Activator programming unit.

† Contributed by FPGA Appilcations Staff, Texas Instruments Incorporated.

Figure 2-15. TI-ALS Viewlogic Design Flow



Note:

The Activator 1 programming unit will interface only with a PC-386DX or PC-486DX and can program one unit at a time. The Activator 2 programming unit will interface with a PC-386DX or PC-486DX, Sun or HP/Apollo workstations and can program up to four units at a time. *Also, the Activator 1 product was discontinued in January 1993. TI-ALS software continues to support this programing unit.*

2.4.2 Symbolic Links

The TI-ALS software searches for executable and design files in the `/usr/als` directory. If you wish to install the TI-ALS software in a directory other than `/usr/als` you must create a symbolic link from `/usr/als` to the directory that contains the TI-ALS software.

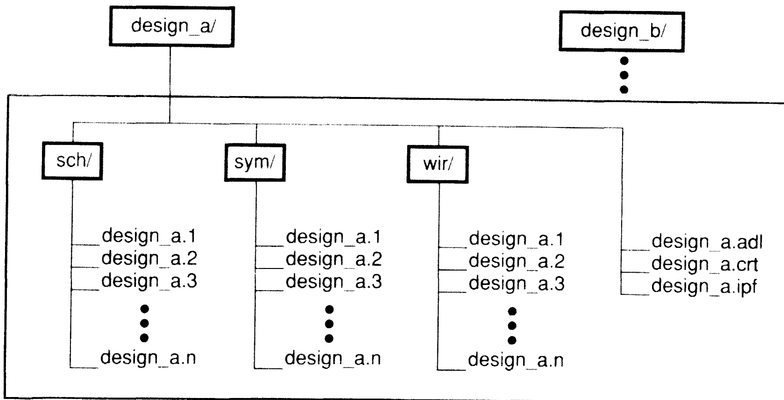
For example, if the TI-ALS software is installed in the `/home1/ti` the following symbolic link must be created:

```
ln -s /home1/ti/als /usr/als
```

2.4.3 Directory Structure

The directory structure when using TI-ALS Viewlogic software is illustrated in Figure 2-16. There is one directory for each design: `design_a` and `design_b` are examples of these directories. The `design_a` directory in Figure 2-16 illustrates what each of these directories will look like. There are three subdirectories under the `design_a` directory: schematic (`/sch`), symbol (`/sym`), and wire (`/wir`). For every page of the design, a file resides in both the `/sch` and `/wir` subdirectories. Each file will have the name of the schematic with the page number as its extension. Similarly, for each symbol, there will be a file in the `/sym` subdirectory. When backing up your design you need to save only the `sch` and `sym` directory files. The `wir` files can be regenerated.

Figure 2-16. TI ALS Viewlogic Directory Structure



2.4.4 TI-ALS/Viewlogic Library Files

The files for the TI-ALS library components are located in:

`/usr/als/lib/wv/act1/cells` (for the TPC10 series family)

`/usr/als/lib/wv/act2/cells` (for the TPC12 series family)

Designs cannot use TPC10 and TPC12 macros simultaneously.

2.4.5 TI-ALS/Viewlogic Simulation Models

The simulation models represent the function of each TI-ALS macro in terms of Viewlogic simulation primitives. The simulation primitives for the TI-ALS library are located in:

`/usr/als/lib/wv/act1/models` (for the TPC10 series family)

`/usr/als/lib/wv/act2/models` (for the TPC12 series family)

2.4.6 Specifying Libraries in the `viewdraw.ini` File (Workview 4.1)

The TI-ALS library directories must be specified at the bottom of the `viewdraw.ini` file as follows:

For TPC10 series:

- DIR [pw].
- DIR [rm] `/usr/als/lib/wv/act1/cells`
- DIR [rm] `/usr/als/lib/wv/act1/models`
- DIR [rm] /*workview_path*/builtin

For TPC12 series:

- DIR [pw].
- DIR [rm] `/usr/als/lib/wv/act2/cells`
- DIR [rm] `/usr/als/lib/wv/act2/models`
- DIR [rm] /*workview_path*/builtin

Where /*workview_path* is the pathname of the Workview directory.

2.4.7 UNIX Environment Requirements

You must alter the UNIX™ `.cshrc` file to add the Workview directory to the search path as well as to set the Workview environment variable by typing the following:

```
setenv WDIR workview_path/standard
```

Where `workview_path` is the pathname of the Workview directory.

2.4.8 Adding Power and Ground

To add power or ground signals in the schematic, use the V_{CC} or GND symbols.

2.4.9 Modifying TI-ALS Soft Macros

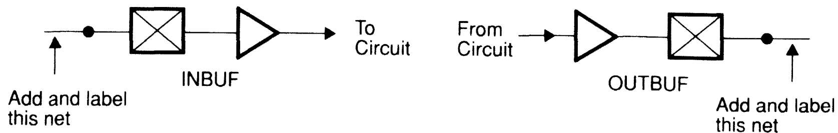
The schematics and symbols of the TI-ALS soft macros can be copied, modified, and saved as user-defined macros by using the Viewlogic `File|WriteTo` menu option. This option will save the schematic and symbol to another file. The new symbol must have the invisible attribute `Level=Soft` removed. Remove the `Level=Soft` attribute as follows:

1) Edit the new symbol.
2) Change all of the attributes to `Visible`.
3) Select and delete the text, `Level=Soft`.
4) Save the edited symbol.

2.4.10 Adding Pins to the Design

Add pins to the design by using the I/O buffer macros, which you should place on the top-level schematic. Then add a net segment to the pad of the I/O buffer and label it, as shown in Figure 2-17. The name of the net attached to the pad corresponds to the net label. This net label is used by TI-ALS for I/O assignment. If all of the I/O buffers are not placed on the top-level schematic, you must create a top-level symbol for the design.

Figure 2-17. Adding Pins to a Viewlogic Design



2.4.11 Top-Level Symbol

For the I/O buffers on lower levels, you must pull the dangling nets up to the top-level schematic. The pin names of the symbol must match the signal names of the dangling nets on the I/O buffer macros.

Note:

Using a top-level symbol may create more work for you because both the symbol and the schematic must change if the I/Os change.

2.4.12 Hierarchy—Sheets Versus Symbols

A multiple page design is composed of more than one `.sch` file. For a multiple page design, each sheet is treated as part of a top-level schematic and is not considered as a hierarchy level. A design using user-defined symbols or soft macro symbols is hierarchical.

2.4.13 Functional Simulation

After you have entered the schematic, a functional simulation netlist, `design_name.vsm`, may be created by selecting `Export|Wirelist|Viewsim` from the Viewlogic menus. The delay through each macro is one nanosecond.

If you are performing a functional simulation following a timing simulation, delete the *`design_name`*`.var` file before performing the functional simulation. This file must be removed because it will modify the intrinsic delay.

2.4.14 Creating the TI-ALS Netlist from the Schematic

After you complete your schematic, exit the Viewlogic software, and at the UNIX system prompt in the design directory type the following command:

vl2adl [fam:<*family_name*>] <*design_name*>

where *<family_name>* is `act1` or `act2`

The **vl2adl** command will use the `.wir` files for the design to create three files: the `.adl` netlist file, and templates for the criticality (`.crt`) and pin (`.ipf`) files. These files will be located in the *`design_name`* directory.

2.4.15 Back Annotating Time Delays

After the `.adl` netlist has been created, use the TI-ALS software to place and route the device. After place and route, invoke the Export program from the UNIX system prompt by typing **export <design_name>**. Use of the Export program will create a timing file, <*design_name*>.dtb, in both the <*design_name*> directory and the Workview directory. The Export program will also create a <*design_name*>.vsm file that contains postroute delay information for Viewsim.

The back annotated delay file contains min/max delay information for the voltage, temperature, and speed grade which are specified in the TI-ALS `Export` menu option. The `viewsim.ini` file should be edited to change the delay typ (typical) command to delay max or delay min.

2.4.16 Viewlogic Anomalies

When selecting the TRIBUFF macro, be sure to type two Fs; otherwise, the Viewlogic TRIBUF macro will be selected.

The TI-ALS system used in conjunction with the Viewlogic tool set provides a powerful desktop FPGA design solution. This fully integrated approach offers you the ability to utilize Viewlogic's popular design entry and design verification tools coupled with the TI-ALS system's automatic device configuration and programming features to quickly turn your design concept into programmed silicon.

2.5 Mentor on Apollo†

2.5.1 Introduction

This section describes the implementation of a video compression algorithm in a TPC10 series FPGA using the Mentor Graphics™ CAD environment on an HP/Apollo workstation. Texas Instruments provides a suite of software tools which interface the Mentor CAD environment to the TI-ALS. The result is the ability to take a Mentor schematic database through to a programmed FPGA.

2.5.2 Design Overview

The design used here to illustrate the Mentor FPGA design flow implements part of a video compression algorithm. A television signal is digitized at 8 bits per pixel with 256 x 256 pixels per frame. The frame is scanned horizontally, one line at a time, and each pixel is processed individually by the system. This application example implements the noise detection part of the compression algorithm.

The intensity level of each pixel is represented by an 8-bit word (Figure 2-18). Each pixel is coded according to its intensity.

Figure 2-18. Pixel Coding

Level	BIT1	BIT0
255		
UPL	1	1
LWL	0	0
0	0	1

The upper-band level (UPL) and lower-band level (LWL) are set by the user. Two bits are used to code each pixel: bit 0 is used to indicate the absence or presence of a pixel, and bit 1 is used to indicate the intensity of the pixel (with 0 representing a white pixel and 1 representing a black pixel).

The noise detector operates on bit 0 of this code and is designed to remove rogue signals which are surrounded to a greater or lesser extent by zeros. Figure 2-19 shows possible pixel patterns in which a present signal (1) is surrounded by a preponderance of absent signals (0).

† Contributed by Katey Derbyshire, FPGA Applications, Texas Instruments Ltd.

Figure 2-19. Possible Noise Detector Patterns

```
a)            b)            c)
   0 0 0         x 0 x         0 0 0
   0 1 0         0 1 0         0 1 1
   0 0 0         x 0 x         1 1 0
```

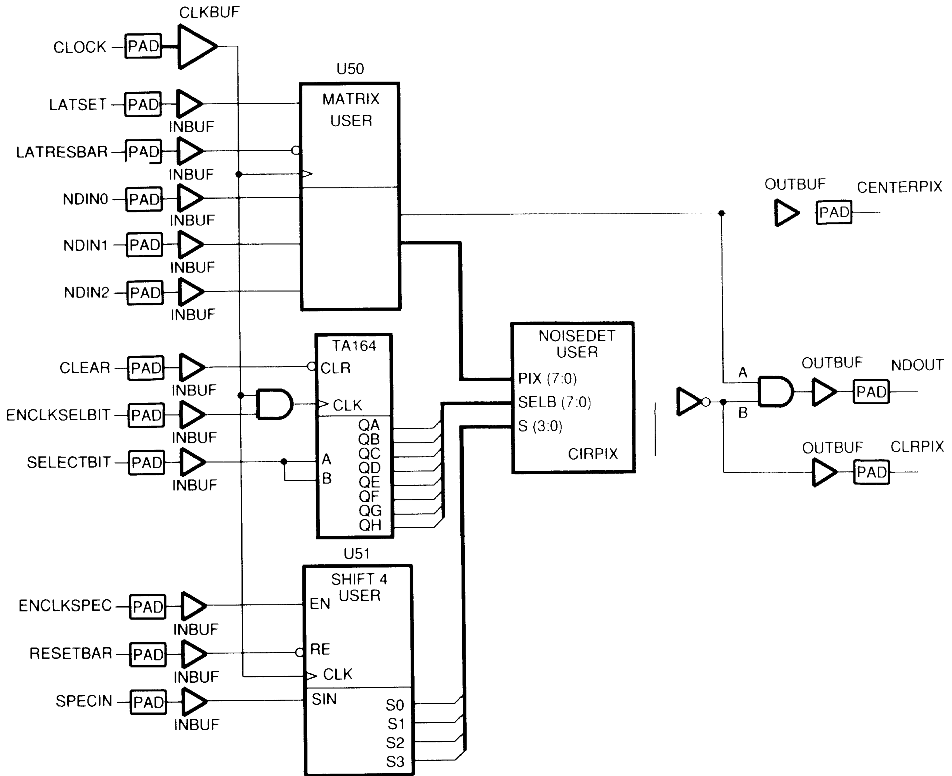
Each is an example of possible rules which identify the center pixel as noise and must be cancelled. The user specifies which of the positions surrounding the signal are to be considered by the noise detector and how many bits must be zero in order to classify the signal as noise and therefore to cancel it. The position of the surrounding bits are numbered 0 to 7 (Figure 2-20).

Figure 2-20. Matrix Numbering

```
0 1 2
3 ? 4
5 6 7
```

The top-level design implementation using the TPC10 series library is shown in Figure 2-21. The positions to be considered by the noise detector are specified on `selectbit`. To classify the center pixel as noise, a specified adjacent pixel must be zero. This number is specified on `specin`.

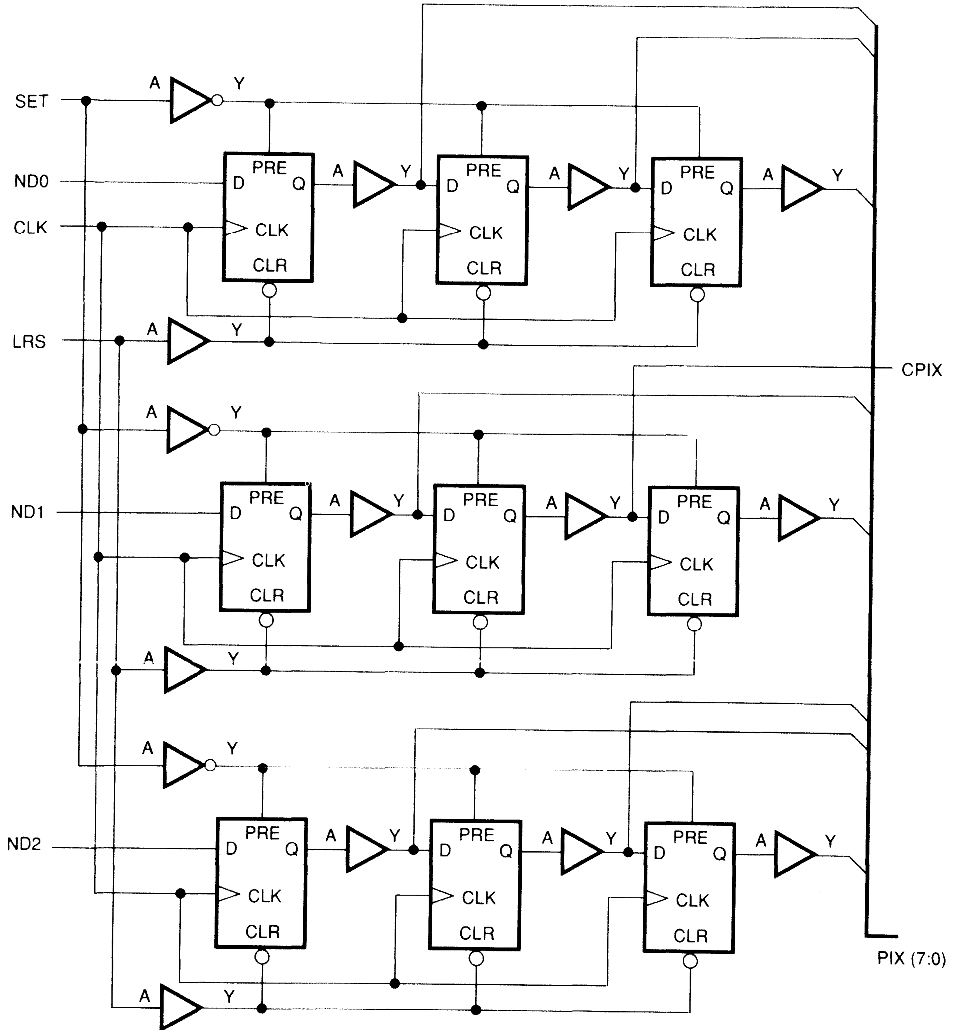
Figure 2-21. Top-Level Schematic



There are three mains blocks in the design: a data management block consisting of various register types, a 3 x 3 matrix of D-type flip-flops, and a summation block consisting of adders of various widths.

Each input to the noise detector (ndin0, ndin1, ndin2) represents one element in a column of the 3 x 3 matrix shown in Figure 2-22. As the image is scanned from left to right the next column of pixels is clocked into the array and the new block is processed by the noise detector.

Figure 2-22. Matrix Schematic



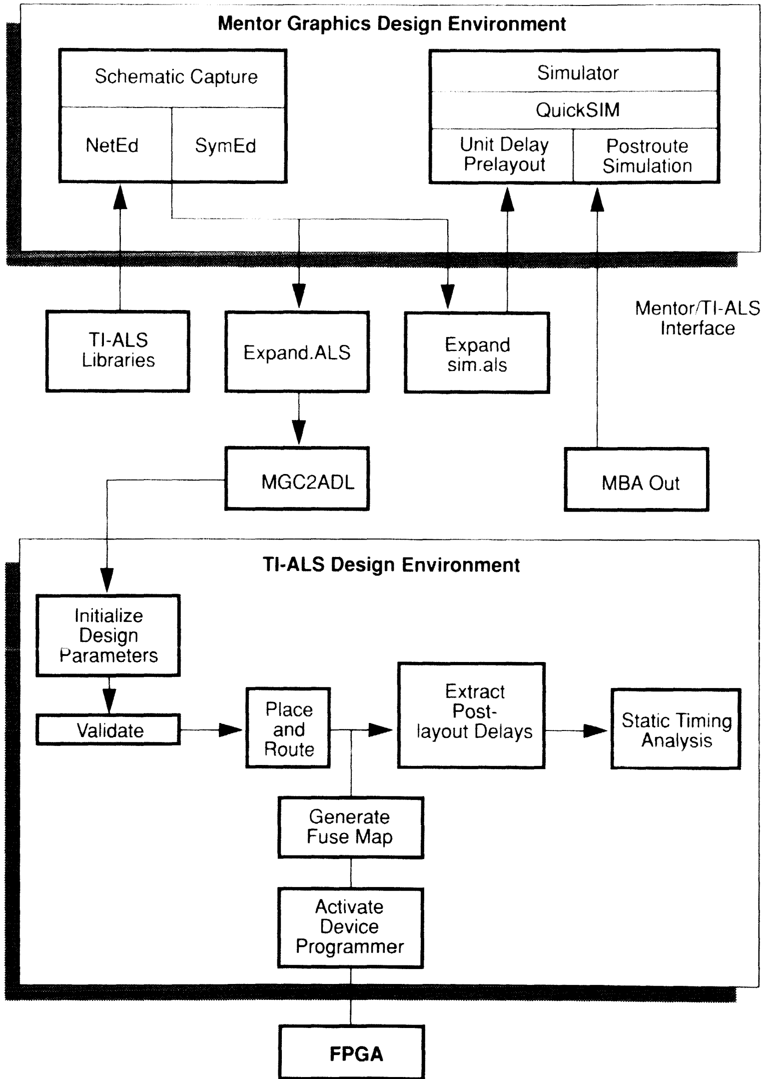
The adder logic counts the number of logical 1s there are in the positions being considered by the noise detector and outputs any number between 0 and 8 depending on the specification. This number is then compared to the control signal and if the number of 1s counted is less than the specification,

clrpix goes high indicating that the center pixel is noise cancelled. The cancellation operation is performed by inverting the comparator output and ANDing it with the original center pixel. The pixel will therefore be removed if clrpix is high.

2.5.3 Design Flow—Schematic Capture

The TI-ALS software is supplied on a standard data cartridge tape, comes with installation instructions, and includes an installation program to set up the TI-ALS environment on an HP/Apollo workstation. The software will run under Aegis™ version 10.1 and above. The path `/user/als/bin` should be added to the search path either in the startup file or in the shell using **`csr -a`** command. The diagram in Figure 2-23 illustrates the complete Mentor design flow.

Figure 2-23. Mentor/TI-ALS FPGA Design Flow



The Mentor schematic capture package is invoked by keying the command **neted** ***<design_name>***. In order to access all the TPC10 series cells and soft macros the FPGA library must be read into the NetEd™ menu window (`READ MEnu /user/als/lib/mgc/act1/cells`). Only one set of libraries should be used in the design, either the 10 or 12 series—they should never be mixed. Alternatively, the HP/Apollo system can be set up to enable custom entry into NetEd. For example a call to `neted.a1000` might automatically set up the TPC10 series library and a call to `neted.a1200`, the TPC12 series library.

Any user-defined soft macros in the design, such as the matrix, shift4, and noise detector blocks, must each have their own symbol generated in SymEd™, the Mentor symbol editor. Each symbol must have a `Level` property assigned to it whose value is `User`. This property is required so that the macro is recognized by the netlist extraction tool as a user-generated macro and not a library cell. The tool will not search the library for the cell but will go down into the macro and search for the FPGA primitives. If the property is added to the initial symbols, ensure that the top-level schematic is updated to incorporate the new property (`Update/Part/Instance`).

The top-level FPGA schematic should contain the necessary input, output, and if used, clock buffers, for the design. It is advisable to label all nets and instances in the schematic to make simulation and debug of the design easier.

2.5.4 Design Verification

The design is simulated using QuickSIM™, the Mentor digital simulator. Prior to simulation the design must be expanded into a single Mentor database. A tool, `expand_sim.als` is provided with the installation for this purpose. The syntax of this tool is:

```
expand_sim.als <design_name>
```

The design is expanded to incorporate all the cell timing attributes for use in the simulator. The simulation database is annotated by default with prelayout unit delays.

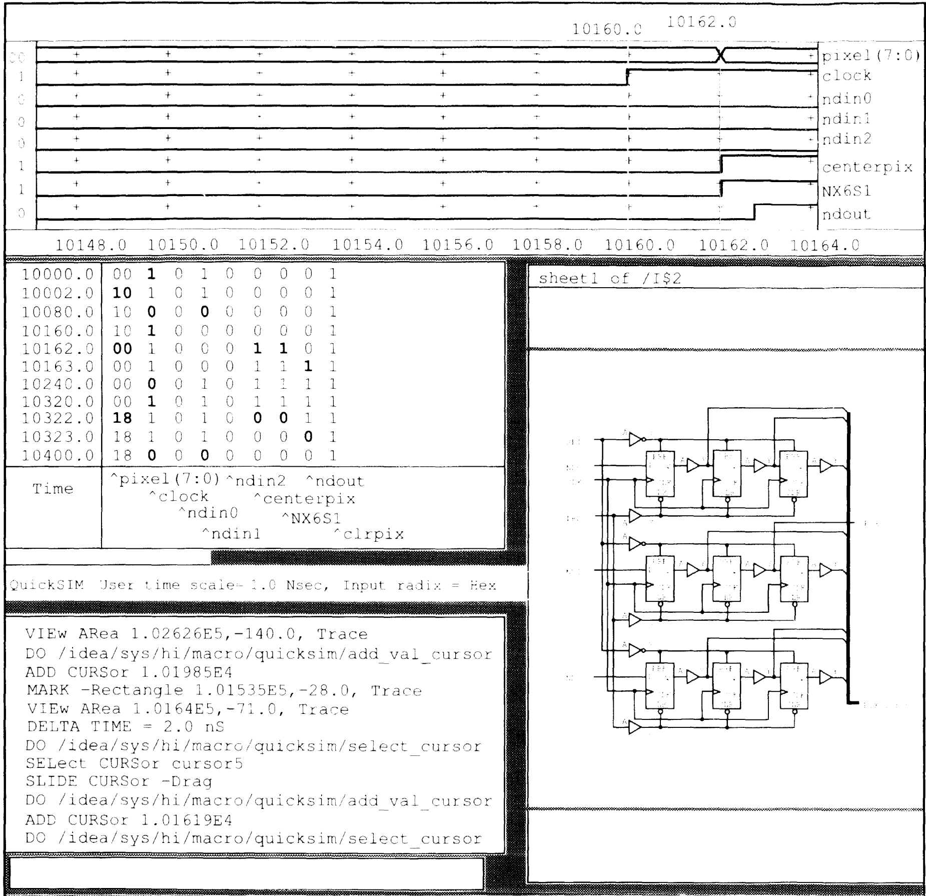
In this application example the Mentor human interface macro language is used to create a simulation command file. Part of this command file is shown in Figure 2-24; initially, the simulation environment is set and then stimulus is applied to the inputs and the functionality of the design tested.

Figure 2-24. Simulation Command File

```
# MENTOR Simulation Command File

# Set up the graphical environment

template FORCe wired

# Set up signals to trace

TRAce pixel (7:0) clock ndin0 ndin1 ndin2 centrepix NX6S1 ndout
clrpix

LIST hex pixel (7:0) clock ndin0 ndin1 ndin2 centrepix NX6S1
ndout clrpix -Change

# Simulate the design

FORCe clear 0

FORCe clear 1 160

FORCe resetbar 0

FORCe resetbar 1 160

FORCe latresbar 0

FORCe latresbar 1 1760

FORCe ndin2 0

FORCe ndin2 1 2560

FORce ndin2 0 2880

FORCe ndin2 1 3200

FORCe ndin2 0 3520

clock period 160

FORCe clock 0 0 -r

run 8000
```

A typical QuickSIM output is shown in Figure 2-25 for this design example. Using the cursors and the QuickSIM delta time utility, the delay from the clock to the output centrepix can be evaluated.

Figure 2-25. Prelayout Simulation Output



The delay is 2 ns, and consists of unit delays through the flip-flop and the buffer. No delay is associated with the output buffers during prelayout simulation.

2.5.5 TI Action Logic System

Once the design has been completed and simulated functionally the ADL netlist is extracted from the Mentor database. This is done using two tools provided with the installation, the Expand and MGC2ADL utilities. The syntax of these tools is:

```
expand.als <design_name>

mgc2adl <design_name>
```

The Mentor Expand utility creates a single design database which contains all the cell interconnects to a model level required by the netlist extraction tool. This program should be run at each level of the design hierarchy. MGC2ADL extracts a text file from the Mentor design database with the suffix `.adl`. This is referred to as the ADL file and is the netlist description of the design which interfaces to the TI-ALS. The design can then be ported to the TI-ALS environment.

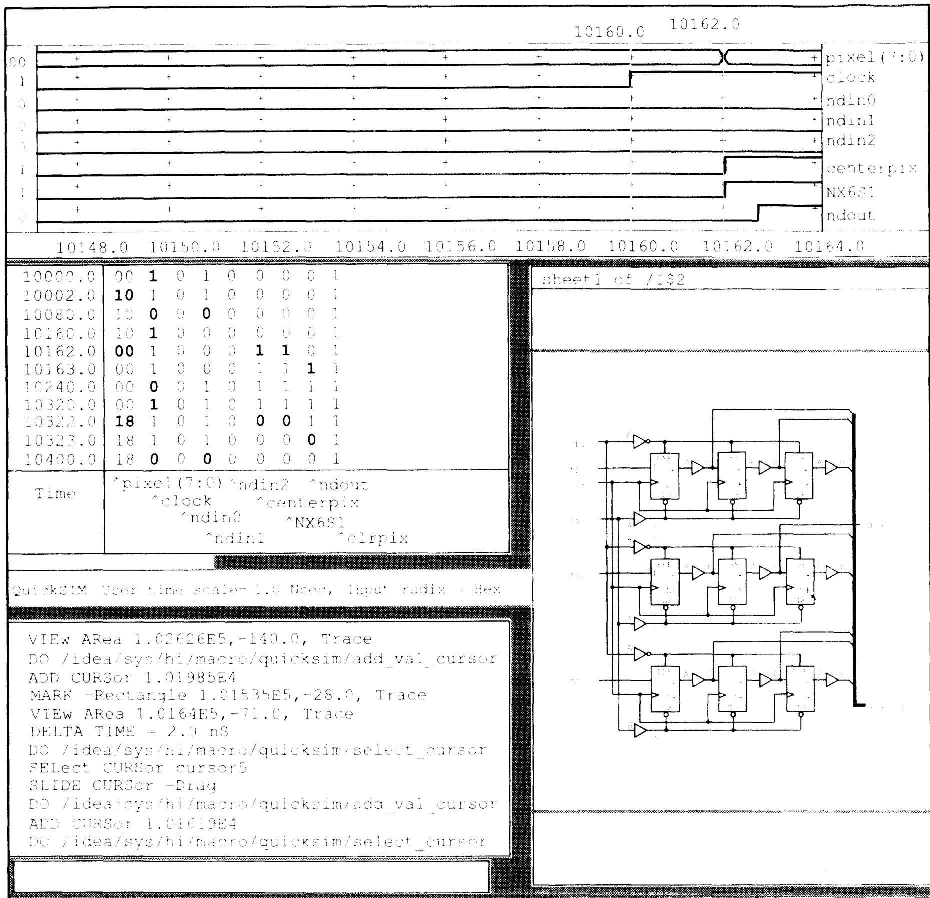
Typing **als <*design_name*>** at the shell prompt of a licensed node invokes the TI-ALS software with the graphics window. The TI-ALS design flow is covered in detail in another section of this handbook. For this specific example the device selected is a TPC1020 in a 44PLCC package. The design is validated and gives an initial module count of 113 logic modules, with 15 I/Os. No errors or warnings are given. The configuration option then automatically assigns pins, (provided manual pin assignment has not already been performed) and places and routes the design.

After place and route, physical timing delay information is extracted from TI-ALS and wire delays are back annotated to QuickSIM. The Export tool option in TI-ALS enables you to set the back annotation environment, temperature, voltage, speed grade, etc. The postlayout delays are held in the delay file (`.del`) and are annotated to the Mentor simulation database using the tool `export.als`. The syntax for this tool is:

```
export.als <design_name>
```

This program calls the delay parameters set up in the TI-ALS environment and the interconnect delays from layout, and automatically annotates the Mentor database with postlayout delays. The design is now ready for postlayout simulation using QuickSIM. Figure 2-26 shows a part of the postlayout simulation, with the clock-to-centrepix delay at 44.8 ns.

Figure 2-26. Postlayout Simulation Output



The TI-ALS software can be run from a node other than the one where the license sits by creeping onto the licensed node and using an Aegis batch program script supplied with the installation. The TI-ALS graphical interface cannot be accessed via a remote node. You must initially work directly at the licensed node in order to set the device and package type, after which the TI-ALS software can be run remotely. The script validates the design, and then, providing that there are no errors, assigns pins, places, routes, optimizes, extracts the delay file, and finally creates the fuse map for the design. This program is called ALSRUN. The syntax for ALSRUN is:

```
alsrun <design_name>
```

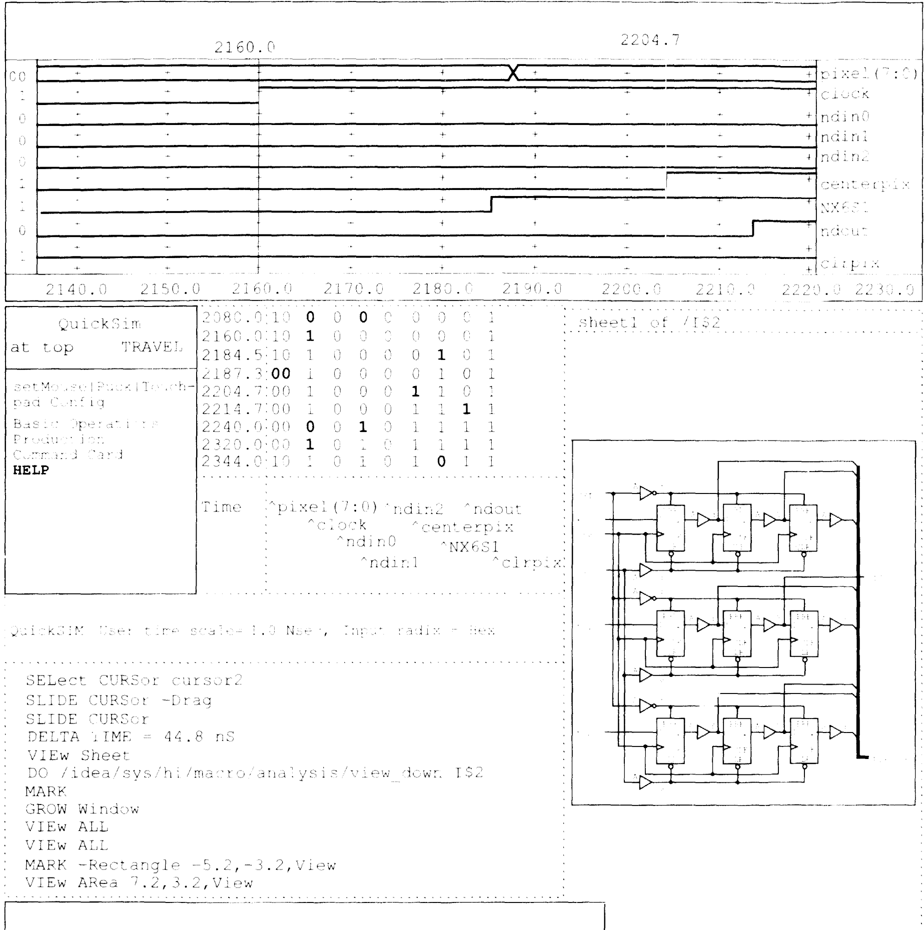
Postlayout delays are back annotated to the design database using the **`export`** command as shown previously. The back annotation environment can be set by selecting various command line options. The syntax for setting up the back annotation environment is:

```
export.alstempr:[ind]voltr:[ind]speed_grade:[std]<design_name>

export.alstempr:[com]voltr:[com]speed_grade:[-1 ]<design_name>

export.alstempr:[mil]voltr:[mil]speed_grade:[-2 ]<design_name>
```

The TI-ALS static timing analyzer can also be run from the command line. For users who are familiar with the Timer syntax this may be a quicker option.

Figure 2-27 shows the clock-to-ndout delay calculated from the QuickSIM environment to be 54.6 ns.

Figure 2-27. Postlayout Delay for Clock-to-ndout



The static timing analyzer verifies these results. Setting the startset to clock and the endset to outpad, the timer gives the clock-to-ndout path as the longest with a delay of 52.3 ns. The timer output for this example is shown in Figure 2-28.

Figure 2-28. Static Timer Analysis Output

```
Working startset 'clock' contains 42 pins

Working endset 'outpad' contains 3 pins

OUTCELL0:PAD        CENTREPIX      OUTBUF

OUTCELL1:PAD        NDOUT          OUTBUF

OUTCELL2:PAD        CLRPIX         OUTBUF

3 pins

 1st longest path to all endpins

Rank  Total  Start pin      First net  End net    End pin

 0    52.3   I$2/I$11:CLK   I$2/N$33   NDOUT      OUTCELL1:PAD

 1    49.1   I$2/I$11:CLK   I$2/N$33   CLRPIX     OUTCELL2:PAD

 2    9.5    I$2/I$9:CLK    I$2/N$35   CENTREPIX  OUTCELL0:PAD

3 pins
```

2.5.6 FPGA Programming

Programmation of the selected device is done using an Activator 2 which will program the TPC10 and TPC12 series FPGAs. The procedure for connection of the Activator 2 to the HP/Apollo depends on the workstation series. The Activator 2 can either be attached to the HP/Apollo SCSI bus or a supplied host adapter inserted into the AT bus slot. If the Activator 2 is being attached to the SCSI bus then it must be assigned a target identification, and the rotary switch on the Activator 2 must be set to this identification.

The Mentor Graphic CAD environment is very useful for capturing and simulating hierarchical FPGA designs when combined with the TI-ALS for targeting FPGA technology.

The Mentor/TI-ALS design environment makes use of the EWS multitasking capabilities by use of batch files for processing when little or no user-interaction is required. The design environment matches one of the most widely used and powerful workstation CAD tools with a new technology that can translate design ideas onto silicon in hours, rather than weeks.

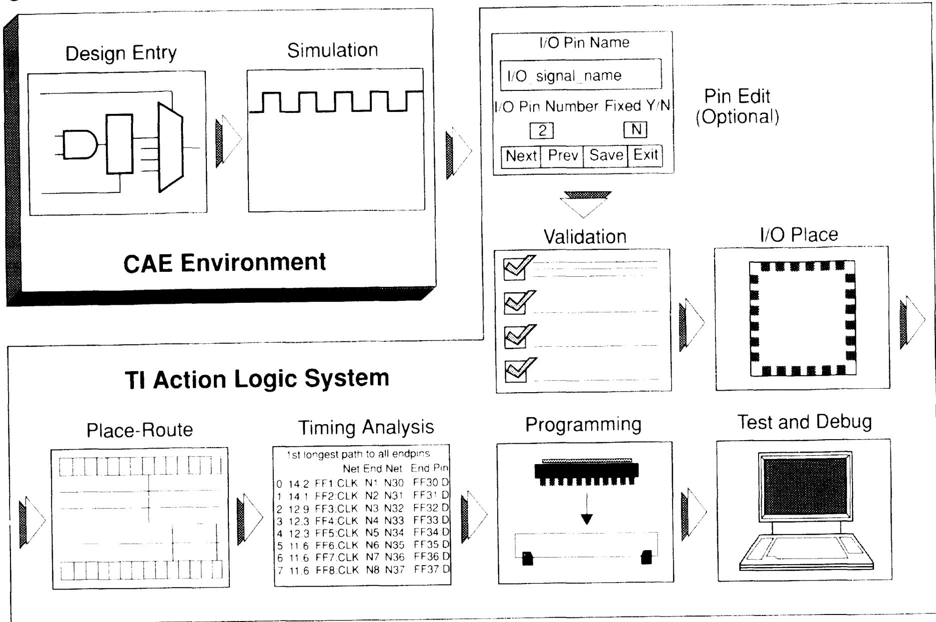
Chapter 3

TI Action Logic System

This chapter discusses the TI Action Logic System (TI-ALS), which consists of software and programming hardware that operates on 386/486 personal computers and HP/Apollo or Sun workstations running popular CAE systems such as Mentor, Valid, Viewlogic, or OrCAD. The TI-ALS enables the quick and easy design of logic solutions using the TI TPC10 series and TPC12 series FPGAs.

3.1 TI Action Logic System Overview†

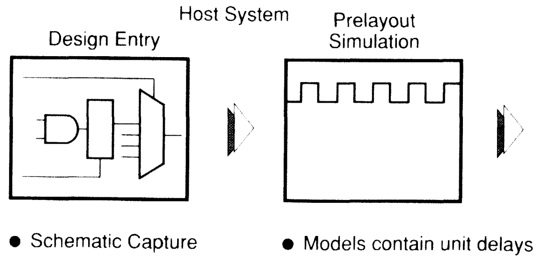
Figure 3-1. CAE Environment and TI Action Logic System



Schematic capture and simulation are performed in the host environment utilizing the TI TPC series macro library which consists of over 500 of the most frequently used logic macros. See Figure 3-2. Both unit delay and postrouting simulations can be carried out within the host environment.

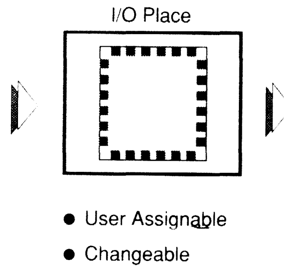
† Contributed by Mohan Maheswaran, Technical Marketing, Texas Instruments Ltd.

Figure 3-2. Schematic Capture and Simulation



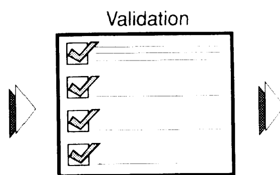
Manual pin assignment allows you to easily assign I/Os to device package pins. See Figure 3-3. Alternatively, they can be automatically assigned by the software. This choice is at your option.

Figure 3-3. Pin Assignment and I/O Placement



Validation (Figure 3-4) is an electronic design rule check that assures design parameters are met so that errors in the schematic can be quickly corrected before proceeding.

Figure 3-4. Validation

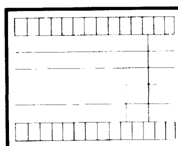


Electronic Design Rule Check

- Module count and utilization
- Fan-out statistics
- Excessive fan-out
- Illegal pin connects
- Unconnected inputs/outputs
- Illegal net names
- Hierarchical errors

Place and route (Figure 3-5) assigns the functions to logic modules and routes the interconnects between logic modules and I/O pins. This is accomplished automatically and is optimized for your design with special emphasis on critical paths. Manual placement of components or routing of interconnects is not necessary. However, utilities provided allow for manual placement of modules. Back annotation exports postlayout delays back to the host environment for device simulation.

Figure 3-5. Place and Route



- Optimizes macro placement
- Picks shortest interconnect net
- 85%-90% array utilization
- 100% automatic
- Typically less than 1 hour runtime
- Calculates segment delays
- Critical path weighting
- Back annotation of wiring delays

Static timing analysis displays the timing characteristics of the FPGA design for inspection of all paths. Refer to Figure 3-6. Postlayout net delays are summarized and automatically ported to the Timer for specific circuit timing analysis.

Figure 3-6. Static Timing Analysis

Static Timing Analyzer

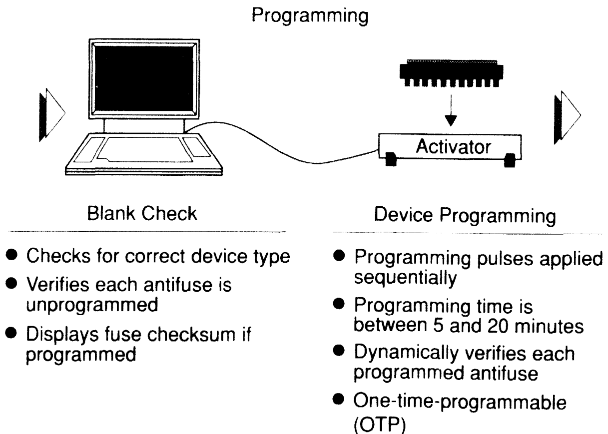
1st longest path to all endpins

		Net	End	Net	End Pin
0	14.2	FF1:CLK	N1	N30	FF30:D
1	14.1	FF2:CLK	N2	N31	FF31:D
2	12.9	FF3:CLK	N3	N32	FF32:D
3	12.3	FF4:CLK	N4	N33	FF33:D
4	12.3	FF5:CLK	N5	N34	FF34:D
5	11 6	FF6:CLK	N6	N35	FF35:D
6	11.6	FF7:CLK	N7	N36	FF36:D
7	11.6	FF8:CLK	N8	N37	FF37:D

- Inspects paths and delays
- Determines path delays
- Optimizes path delays
- Annotates critical paths
- Identifies worst-case path for any pin
- Textual format
- Back annotation of wiring delays

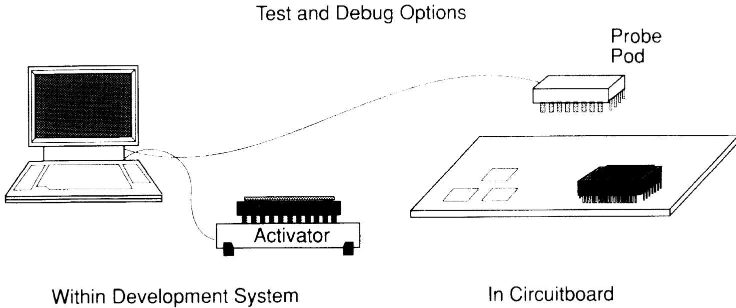
As shown in Figure 3-7, programming is made easy by using the TI Activator. The TI-ALS reads the fuse file and the Activator hardware applies programming pulses in sequence. The target antifuse is verified and a check of surrounding antifuses assures correct device functionality.

Figure 3-7. Programming



Testing and debugging permit verification of a device in the development system environment or target system. One hundred percent observability of all on-chip functions is provided without generating test vectors by using two built-in test pins to address multiple nodes simultaneously. Refer to Figure 3-8. These two built-in test pins can later be used as I/Os. In-circuit testing and debugging are accomplished using an Actionprobe™ which can address any internal node to determine its logical state. Programming security antifuses protects against reverse engineering.

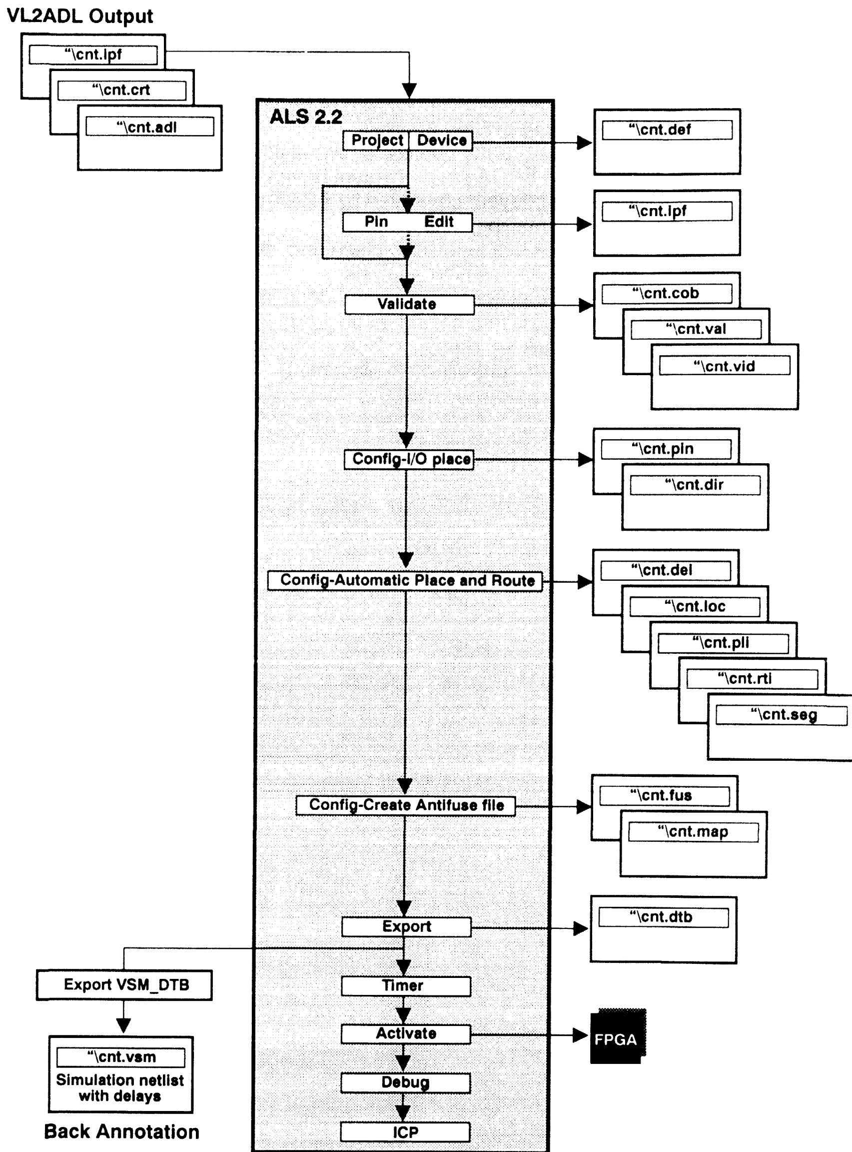
Figure 3-8. Test and Debug



- All internal nodes are 100% observable
- User-defined stimulus vectors or *in-circuit* stimulus can be applied
- Print node analysis to TI-ALS screen or file
- Probe pods allow in-circuit board oscilloscope monitoring of internal nodes
- You can interactively test a programmed device
- You can change stimulus vectors as you go

The complete TI-ALS design flow and associated file structure is shown in Figure 3-9. An example design called `cnt` is used to illustrate filenames. This example flow starts with the ADL file having been produced using Viewlogic-created schematics; consequently, the VL2ADL utility produced the necessary input files (`.lpf`, `.crt`, `.adl`).

Figure 3-9. TI-ALS Design Flow



The following is a list of some of the key features and benefits of the TI-ALS.

- Uses familiar CAE environment (Viewlogic, OrCAD, Mentor, Valid)
- Completes in under 25 minutes for 1200 gates
- Summarizes postlayout net delays
- Allows analysis *in-circuitboard* or *in-programmer box*
- Simplifies in-system analysis
- Provides accurate postlayout simulation
- Allows for FPGA pin placement driven by board-level constraints rather than board layout controlled by FPGA constraints
- Provides multiple netlist inputs
- Completes fast automatic place and route
- Provides static timing analysis
- Provides functional testing and debugging
- Allows 100% internal signal observability
- Allows back annotation to host environment
- Allows user-controlled pin placement

3.2 Configuring Workview for FPGA Libraries†

3.2.1 Viewdraw Initialization File

A Viewdraw initialization file, `viewdraw.ini`, is located in the directory `\workview\standard`. To configure your Workview environment for *TPC10* series FPGA designs, the `viewdraw.ini` file should contain the following lines at the end of the file.

```
dir 11 \als\lib\a1000\cells

dir 12 \als\lib\a1000\models

dir 13 \workview\builtin
```

To configure your Workview environment for *TPC12* series FPGA designs, the `viewdraw.ini` file should contain the following lines at the end of the file.

```
dir 11 \als\lib\a1200\cells

dir 12 \als\lib\a1200\models

dir 13 \workview\builtin
```

A `viewdraw.ini` file is supplied with both the TPC10 series library and the TPC12 series library. The `viewdraw.ini` file supplied with the TPC10 series library is located in the directory `\als\lib\a1000` and the `viewdraw.ini` file supplied with the TPC12 series library is located in the directory `\als\lib\a1200`. To configure your Workview environment for TPC10 series FPGA designs, the `viewdraw.ini` file located in the directory `\workview\standard` should be replaced with the `viewdraw.ini` file located in the directory `\als\lib\a1000`. To configure your Workview environment for TPC12 series FPGA designs, the `viewdraw.ini` file located in the directory `\workview\standard` should be replaced with the `viewdraw.ini` file located in the directory `\als\lib\a1200`.

3.2.2 Default Design Directory

A design database is required for your design files created and used by the TI-ALS. The default design directory is `\designs`. For a design named `counter`, the TI-ALS files created will be stored in the directory `\designs\counter`. The TI-ALS documentation assumes that you are using the default database. According to the installation notes, your system

† Contributed by Glenn Bigger, FPGA Applications, Texas Instruments Incorporated.

will be configured for the default database. However, if you use the Viewfile option when designing with the Viewlogic Workview software, Viewfile will create a different directory structure that will not be compatible with the default system configuration. If you want to use the Viewfile option, please consult the documentation on using Viewfile with TI-ALS and configure your system accordingly.

3.2.3 Component Selection

The TI-ALS supports dialog box library selection which is standard with Workview 4.1 software. The `viewdraw.ini` file supplied with both the TPC10 series library and the TPC12 series Library has the default setting for the dialog box as `OFF`. If you want to use the dialog box to select components when using Viewdraw, you can change the default setting by editing the `viewdraw.ini` file and changing the **dboxon** command from **dboxon 0** to **dboxon 1**. The dialog box can also be enabled/disabled after invoking the Workview software. To enable/disable the dialog box:

1) Invoke the Workview software and execute the menu options `Window Open Viewdraw Schematic` to open an active window.

2) Execute the options `Change Display Params`. The Viewdraw parameters selections will appear in the upper right corner of the display.

3) Move the cursor to the `Dboxon ON/OFF` box. Press the left mouse button then use the backspace to delete the current selection. Type either `ON` or `OFF` and then press the left mouse button. Next move the cursor to the `Saveini` box and then press the left mouse button to save the changes to the `viewdraw.ini` file in the `\workview\standard` directory.

If the dialog box is enabled, components can be added to your schematic through the following steps:

1) Execute the options `Add Comp` and then press the middle mouse button. The dialog box will appear in the middle of the display.

2) Select `DIR` to display the directory selections. Select the library directory. For the TPC10 series library, the directory is `\als\lib\a1000\cells` and for the TPC12 series library, the directory is `\als\lib\a1200\cells`. The FPGA component dialog box will be displayed with the different components; select the desired component.

3.3 Customizing Viewlogic Menus for the TI-ALS†

The design environment for the Texas Instruments FPGA on a PC-386/486 consists of two parts: the TI-ALS and the Viewlogic Workview. TI-ALS is used for FPGA-specific tasks like validation, place and route, static timing analysis, etc. Viewlogic Workview is a set of CAE tools for schematic entry and simulation.

TI-ALS and Workview normally function as two independent software packages with different user interfaces. This section shows how to use the open Workview menu structure to integrate TI-ALS functions or any other MS-DOS command into the Workview menu panels so that these functions can be selected directly from the menu.

3.3.1 Workview Menu Structure

The structure of Workview menus is specified in menu source files which are stored under the subdirectory `\workview\standard`. You can edit these ASCII files to customize the menus for a specific application environment. There is a set of menu source files for each graphic adapter. The files with the extension `.mn1` are for the EGA adapter, `.mn5` files are for CGA, and `.mn8` files are for VGA.

There are menu source files for Utils, Viewdraw, Viewfile, Viewsim, Viewsys, Viewtext, Viewwave, and Workview. When Workview is invoked the first time after the installation, Workview compiles the menu source files into a binary file `menus8.m32` which is later used to configure the menus every time Workview is invoked. Therefore, if the menu source files are modified then the `menus8.m32` file must be deleted or renamed (as a backup) so that it can be recompiled. Otherwise, the modifications will not be effective.

3.3.2 Customizing Workview Menus for the TI-ALS

Each Workview menu source file can consist of several submenus. A submenu defines a menu level and contains source code for the screen coordinates, menu text, and command syntax. Assuming you want to add the following functions into a submenu of the original Workview menu panel:

- `VLITPC10`

  Create ADL netlist for TPC10 series from WIR netlist.
- `VLITPC12`

  Create ADL netlist for TPC12 series from WIR netlist.

† Contributed by Dung Tu, Technical Marketing, Texas Instruments Deutschland GmbH.

- ALS

  Invoke TI-ALS from within Workview.
- ALSRUN

  Invoke a batch file to run a complete TI-ALS flow including back annotation but without fuse generation.
- VSM_DTB

  Back annotate the delays to Viewsim.
- NC

  Invoke Norton Commander™.

These functions can be added to any submenu, such as the `Export` submenu.

Figure 3-10 shows the `Main` menu and the `Export` submenu before and after the modifications.

Figure 3-10. Workview Main Menu and Export Submenu

```
Main menu          Submenu          New submenu

Export             Export
------------------------------------------------------

Window             Wirelist         Wirelist
View               Check            Check
Add                                 VLITPC10 ---
Level                               VLITPC12
Buffer                              ALS          New functions
File                                ALSRUN       |
Change                              VSM_DTB      |
Export <---                         NC        ---
Xform
Select
Delete
Move
Copy
Plot
Undo
Info
```

```
Bye
```

Because both `Export Wirelist` and `Export Check` in the original `Export` submenu do something with a schematic, the `Export` submenu is specified in the Viewdraw menu source file which is the `viewdraw.mn8` for a VGA graphic adapter. Using a text editor to look at the `viewdraw.mn8` file you can find the specification for the `Export` submenu as shown in Figure 3-11.

Figure 3-11. Export Submenu in the Original `viewdraw.mn8` *File*

```
q_export
80 22 {
Wirelist &q_wirelist
Check check
}
.
.
```

Figure 3-12 shows the modified `Export` submenu incorporating the new functions.

Figure 3-12. Modified Export Submenu in `viewdraw.mn8` *File*

```
.
.
q_export
80 22 {
Wirelist &q_wirelist
Check check
VLITPC10 "system vlitpc10 $project"
VLITPC12 "system vlitpc12 $project"
ALS      "system als      $project"
ALSRUN   "system alsrun1  $project"
VSM_DTB  "system export1  $project"
NORTON   "system nc"
}
.
.
```

Looking at the `VLITPC10` line as an example, the syntax is as follows:

- `VLITPC10`

  The name of the function as it will appear on the new menu.
- `system`

  Workview DOS interface call.
- `vlitpc10`

  The name of the batch file to make an ADL netlist for the TPC10 series.
- `$project`

  The project name which will be passed as parameter to the batch file `vlitpc10.bat`.

In the TI-ALS version 2.2 which supports the TPC10, TPC12, and TPC14 series, the `vlitpc10.bat` file looks as follows:

```
vl2adl fam:act1 %1
```

`fam:act1` specifies the TPC10 series as the device family.

`%1` will be replaced by the value of `$project`, which is the name of the current project.

If you do the following steps, which are recommended for creating a new project, then the correct project name will be passed automatically to `$project`.

1) Use `Window, Open, Viewfile, Project, Create` to create a new project.

2) Use `Set, Project` to make the new project to the current project.

3) Use `Window, Open, Viewdraw` and enter a schematic name.

It is mandatory to use the same name for the top-level schematic as for the project name. That means if you have used Viewfile to create a project `top`, then your top-level schematic name must be `top`.

Following are the batch files for the other menu items. All batch files are stored under `\als\bin`.

Figure 3-13. Various Batch Files

```
Menu Item    Batch File
--------------------------------------------------------------
VLITPC12        vlitpc12.bat:
                vl2adl fam:act2 %1
```

ALS — Original `als.bat` file which is included in TI-ALS package.

```
VSM_DTB         export1.bat:
                export tempr:com voltr:com speed_grade:std %1
```

Back annotate the delays to Viewsim for commercial temperature range, commercial voltage range, and standard speed grade.

```
ALSRUN          alsrun1.bat
```

A modified version of the original ALSRUN which does not generate the fuse map but includes the back annotation of the delays to Viewsim.

```
@echo off
if x%1x == xx goto usage
validate %1
ioplace %1
place %1
route %1
xtract %1
export1 %1
goto exit
:usage
echo Please specify the design name.
echo    alsrun design_name
:exit
```

After modifying the Viewdraw menu source file and adding the batch files, the last step is to rename the file `menus8.m32` to a backup file. When you restart Workview you can see the message that Workview recompiles the menu files and when you click `Export` you see the new menu items.

If you open a schematic window and enter a schematic name and click `Export, VLITPC10`, the **makeadl** command function is invoked and an ADL netlist for the schematic is generated using TPC10 as the device family. When this process is completed and you press a key you will be back in the Workview environment.

The `ALSRUN` submenu is useful if a schematic is modified. After changing the schematic and generating a new ADL netlist, clicking `ALSRUN` will invoke the complete TI-ALS flow and back annotate the delays to Viewsim without generating the fuse file.

Customizing Workview menus to incorporate the TI-ALS functions or any useful MS-DOS utility releases you from entering batch commands with long input parameters and helps improve your productivity.

3.4 Understanding the ADL Netlist†

3.4.1 Introduction

3.4.1.1 The ADL Netlist

The ADL netlist is the format by which information is passed from a host CAE system to the TI-ALS. Its text consists entirely of readable ASCII characters. It contains information that describes a design's connectivity, I/O, and components.

Although it is not necessary to understand the ADL netlist to perform FPGA design, it can be a useful tool for debugging or documenting a design. Sometimes it is easier to find circuit information or a design flaw when it is expressed textually instead of graphically. With this point in mind, a simple design example and its corresponding ADL netlist have been prepared to illustrate some features of and to provide familiarity with the ADL file. For this section the Viewlogic-compatible utility (VL2ADL) was used. The utility name may vary for other CAE environments.

3.4.1.2 ADL Netlist Generation

The ADL file is created from the host CAE database by the VL2ADL program. The correct syntax that you type to invoke VL2ADL is:

vl2adl [fam:*<family_name>*] *<design_name>*

where *<family_name>* is `act1` or `act2`. Use `act1` for TPC10 series and `act2` for TPC12 series designs.

3.4.1.3 ADL Netlist Location

The output is then placed in the design directory and the full pathname to the file is:

C:\designs\<design_name>\<design_name>.adl

3.4.2 Example Design and Netlist

3.4.2.1 Example Design Description

A simple design that will be analyzed is shown in Figure 3-14 and Figure 3-15. Figure 3-14 contains the top-level schematic of this hierarchical example. Two blocks are used to represent more detailed circuitry at a lower

† Contributed by Joel S. Lason P.E., FPGA Applications, Texas Instruments Incorporated.

level. In addition, there is an AND gate that is a primitive and therefore no lower level drawing exists for this symbol.

Figure 3-14. AP1 Top-Level Schematic

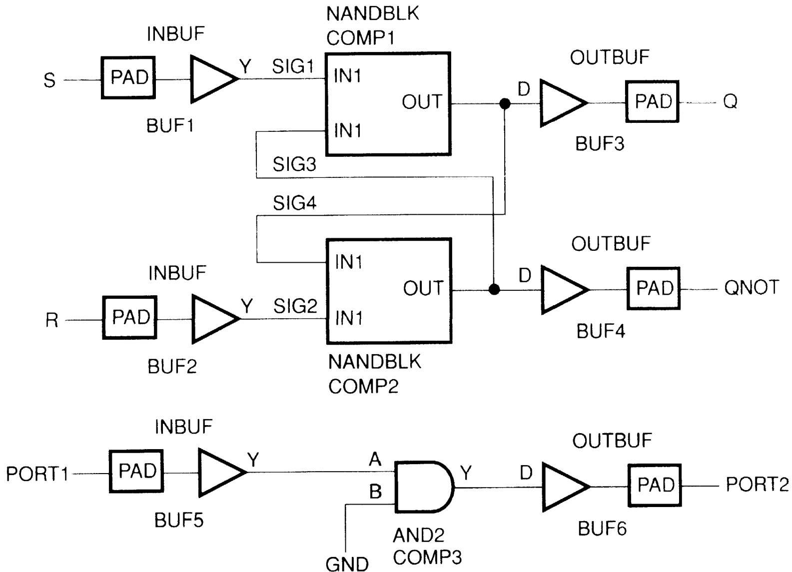
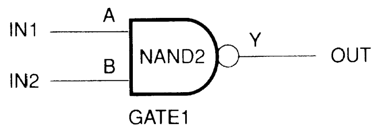


Figure 3-15 shows that the only logic underneath the NANDBLK symbol is a NAND gate. This was done to illustrate the structure of the ADL netlist for a simple hierarchical design.

Figure 3-15. NANDBLK Sublevel Schematic



3.4.2.2 Example ADL Netlist Listing

The netlist for this particular design is shown here in Figure 3-16 and will now be examined. Line numbers have been added to aid in the discussion of the file. They do not exist in the actual file. This example is from a design called `DEMO21`.

Figure 3-16. ADL Netlist Listing

```
1 ; HEADER
2 ; FILEID ADL /designs/AP1/AP1.adl c6e91786
3 ; CHECKSUM c6e91786
4 ; PROGRAM v12adl
5 ; VERSION 2.2
6 ; VAR DEFSYS /als/data/system.def
7 ; VAR DEFUSR /alsuser/jdoe/jdoe.def
8 ; VAR DEFDES /designs/DEMO21/DEMO21.def
9 ; VAR DEFDES /designs/AP1/AP1.def
10 ; VAR design AP1
11 ; VAR FAM ACT1
12 ; VAR ALLFAMS ACT1 ACT2
13 ; VAR ADLIB /als/data/a1000/adl04.lib
14 ; VAR DESDIR /designs/AP1
15 ; VAR AAL /designs/AP1/AP1.aal
16 ; VAR ADL /designs/AP1/AP1.adl
17 ; VAR DELETEINSTPROPS LEVEL
18 ; VAR MSGINDEX /als/data/als21.idx
19 ; VAR MSGTEXT /als/data/als21.txt
20 ; VAR PERMITBADNAMES <NOT-SET>
21 ; ENDHEADER
22 DEF NANDBLK; IN1, IN2, OUT.
23 USE ADLIB:NAND2; GATE1.
24 NET IN1; IN1, GATE1:A.
25 NET IN2; IN2, GATE1:B.
26 NET OUT; OUT, GATE1:Y.
27 END.
```

```
28
29 DEF AP1; S, R, Q, QNOT, PORT1, PORT2.
30 USE ADLIB:OUTBUF; BUF6.
31 USE ADLIB:INBUF; BUF5.
32 USE ADLIB:INBUF; BUF1.
33 USE ADLIB:INBUF; BUF2.
34 USE ADLIB:AND2; COMP3.
35 USE NANDBLK; COMP1.
36 USE NANDBLK; COMP2.
37 USE ADLIB:OUTBUF; BUF4.
38 USE ADLIB:OUTBUF; BUF3.
39 NET S; S, BUF1:PAD.
40 NET R; R, BUF2:PAD.
41 NET Q; Q, BUF3:PAD.
42 NET QNOT; QNOT, BUF4:PAD.
43 NET PORT1; PORT1, BUF5:PAD.
44 NET PORT2; PORT2, BUF6:PAD.
45 NET $1N21; COMP3:A, BUF5:Y.
46 NET $1N27; BUF6:D, COMP3:Y.
47 NET GND; COMP3:B; GLOBAL, POWER:GND.
48 NET SIG1; COMP1:IN1, BUF1:Y.
49 NET SIG2; COMP2:IN2, BUF2:Y.
50 NET SIG3; BUF4:D, COMP1:IN2, COMP2:OUT.
51 NET SIG4; BUF3:D, COMP1:OUT, COMP2:IN1.
52 END.
```

Because the design is all ASCII it is simple to read and understand. Just by looking at this file you can find several obvious types of lines which begin with DEF, USE, NET, END, and the semicolon character (;).

- Comment lines

  Lines 1–21 begin with a semicolon and all of these lines are comment lines. They contain information about the file and the conditions under which it is generated.

- `DEF` lines

  The `DEF` lines define a functional block of the design. The first `DEF` line is line 22. This line begins the definition of the block that represents the NAND gates. The next `DEF` line is line 29, which defines the top level of the schematic, `AP1`. For simplicity these are all the blocks defined in this example, yet any number of blocks could be used.

  `DEF` lines also call out all the I/O signals for a given functional block of the design. Line 29 shows that the I/O signals defined for the AP1 block are `S`, `R`, `Q`, `QNOT`, `PORT1`, and `PORT2`. These signals connect to various signal nets in the design.

- `USE` lines

  Line 23 is the first example of a `USE` line, which tells the TI-ALS which functional blocks are used in this section. On line 23 the `USE` line is specifying that a NAND2 function from the standard TPC library is used and that its instance name is `GATE1`. Other `USE` lines in the design call in I/O buffers and `AND2` standard library functions.

  Hierarchy is illustrated by the `USE` lines 35 and 36. These lines specify that the circuit defined above, NANDBLK, is called into the `AP1 DEF` block. This differs from the other `USE` lines which call standard library components. The two uses of the NANDBLK definition are assigned to circuit instances `COMP1` and `COMP2`.

- `NET` lines

  `NET` lines define connections between components. Line 48 shows that a signal named `SIG1` is connected to the circuit instance `COMP1` and pin `IN1` on `COMP1`. `SIG1` is also connected to instance `BUF1` and its output pin `Y`.

  Line 45 shows what happens if all nets are not explicitly labeled. The system creates a default name for the `NET` and in this case the `NET` name is `$1N21`. This is harder to use for debugging because it contains less intuitive information.

  A special type of signal is illustrated in line 47. This signal is a global signal as shown by the `GLOBAL` modifier on the `POWER:GND` pin definition.

- `END` lines

  The last line to discuss is the `END` line. Its simple function is to terminate functional block definitions called out by the `DEF` lines.

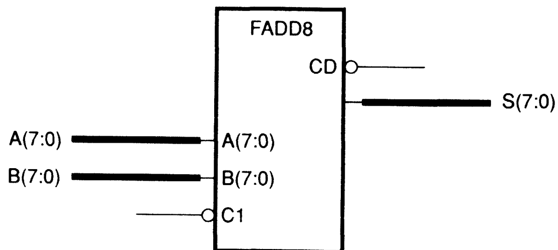
The ADL netlist is easy to use and provides quick design insight. Checking the netlist is quicker in some instances than invoking a graphical schematic capture package, and it presents the design in a different perspective. Sometimes a different view is exactly what is needed to eliminate confusion. Because of its simplicity and utility it is a worthwhile file to understand.

3.5 Back Annotated Simulation for FPGAs†

3.5.1 Introduction

In Section 3.5 the use of the TI-ALS in conjunction with the Viewlogic environment is presented. You can use TI-ALS and Viewlogic to produce actual timing delays associated with an FPGA design and can subsequently carry out a postrouting simulation. The process of translating the data generated from the TI-ALS environment to the CAE design environment is referred to as back annotation (i.e., postlayout delay information is back annotated to the CAE simulator). The 8-bit full adder is used as a design example (Figure 3-17) to describe the process of back annotating. The ability to back annotate is a very powerful feature of the TI-ALS which, when used with a good simulator, can verify both functionality and timing after routing the design.

Figure 3-17. An 8-Bit Full Adder Using Soft Macro FADD8



3.5.2 Prelayout Delay Estimations

The first step in the design process is to carry out a functional, or unit delay, simulation which will indicate whether the design is behaving as expected. Once this is established you can begin to take a closer look at the timing of the circuit.

Essentially there are two ways, prior to routing the circuit, that the timings associated with a design can be estimated. Firstly, a prelayout delay estimation can be made using the statistically estimated delays in the data sheet and secondly, a prelayout delay simulation can be carried out using scaling factors in the Viewlogic environment. The combination of both should give you a good idea of whether the FPGA is going to meet design requirements and whether there are any areas that may need more careful attention before the design is routed. This is an important part of the design

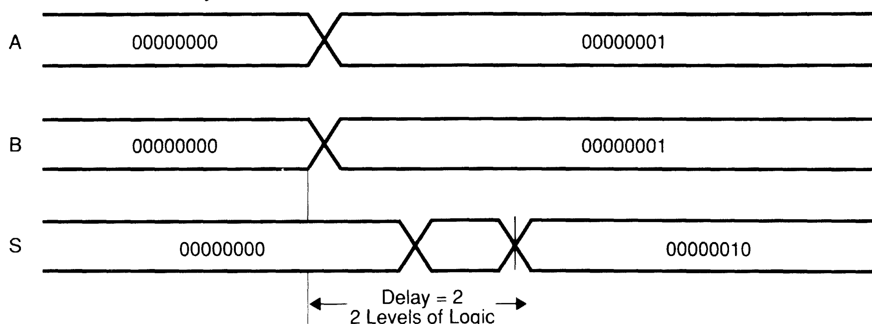
† Contributed by Mohan Maheswaran, Technical Marketing, Texas Instruments Ltd.

process as timings can be influenced significantly by the sensible assignment of pins and criticality.

3.5.3 Unit Delay Simulation

The delay of any path is accumulated from a combination of a combinational block, a sequential block, an input and output delay, and routing delays. For the calculation of the delays through any combinational or sequential block, the number of logic levels that make up the block must first be determined. This information can be obtained from the FPGA data sheet. For example, the 8-bit full adder has two logic levels. For the purposes of functional simulation, the simulator assigns one unit delay for each level of logic in the design. Thus an 8-bit adder using the soft macro FADD8 would have a delay of two units as shown in Figure 3-18.

Figure 3-18. Unit Delay Simulation of 00000001 + 00000001

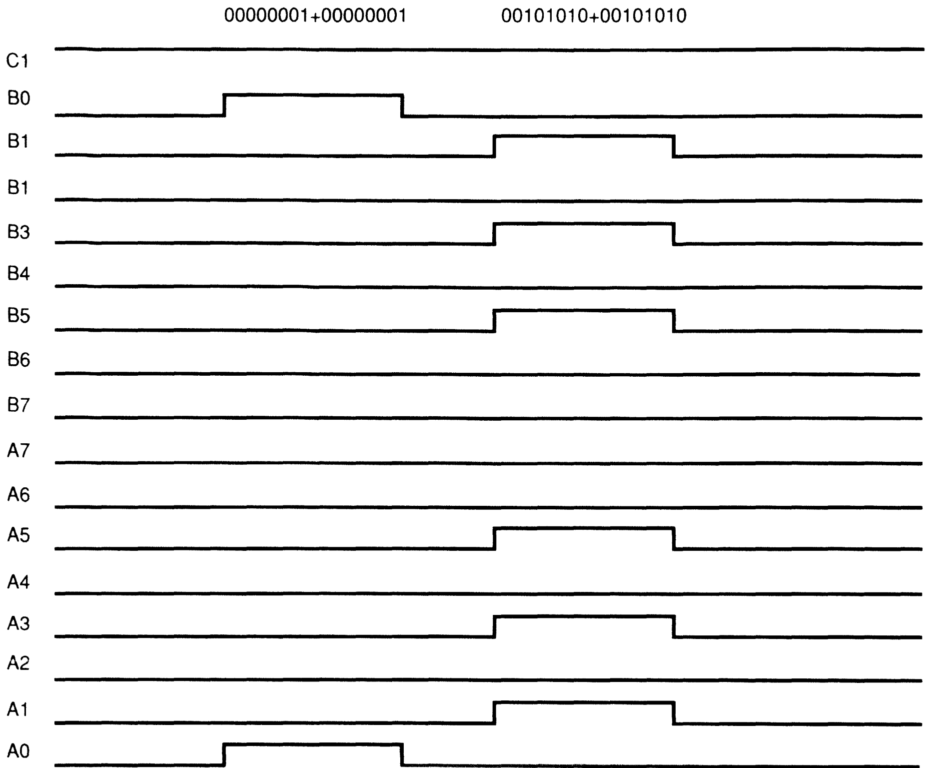


3.5.4 Postrouting Simulation

After completing the routing of the design and exporting the delays from the TI-ALS environment to the simulation environment it is now possible to carry out a postrouting simulation with the back annotated delays.

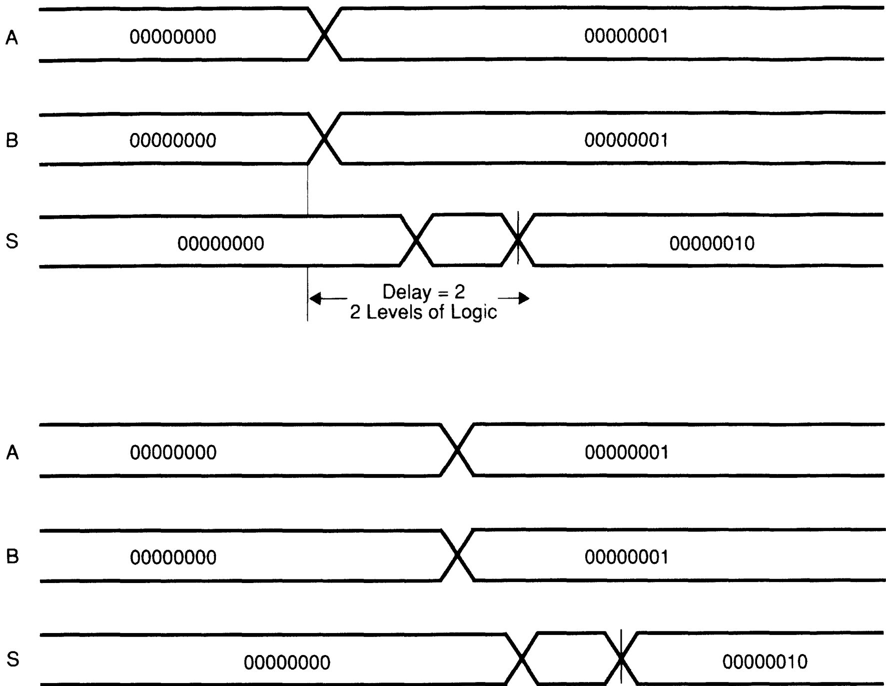
The input stimulus used for simulation of the 8-bit full adder is shown in Figure 3-19. The task is to add 00000001 and 00000001 together as fast as possible and then add 00101010 and 00101010 together as fast as possible.

Figure 3-19. Input Stimuli to Full Adder



The result for the first 8-bit addition is shown in Figure 3-20. The 8-bit addition takes just 16.6 ns representing an adding operating frequency of above 60 MHz. This is the typical performance of many complex functions using the TI FPGAs. The second addition occurred in a comparable time.

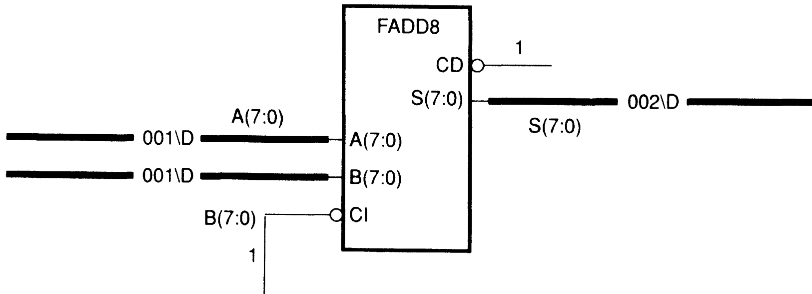
Figure 3-20. Postrouting Simulation of 00000001 + 00000001



Having carried out a postrouting simulation to prove the functionality of the design and assess the timing performance, it is now good practice to try to simulate at best and worst-case delays. This is possible using Viewsim's ability to adjust all the delays by some defined factor. The scaling factors that would give best and worst-case timings of the circuit can be found in the appropriate TI FPGA data sheet.

Debugging timing problems within a circuit can be difficult if the development tools do not facilitate this process in any way. Fortunately, Viewsim has a very useful feature which allows logic levels following simulation to be seen on a schematic. Thus the simulator can be set to break on a count or an event and the resulting logic levels at the break point can be seen on the schematic (Figure 3-21). By using this very useful feature it is possible to carry out a debug of the circuit without programming any devices. This is made even more useful because of the ability to *push down* through the schematic hierarchy and see the logic levels at the logic-module level. Thus, back annotation and postrouting simulation are a key part of the FPGA design flow.

Figure 3-21. Postrouting Simulation Results Shown on Schematic



In summary, delays can be back annotated from the TI-ALS environment to the Viewlogic environment and a postrouting simulation carried out. The complete FPGA system offers you the ability to carry out timing simulations as well as purely functional simulations. This capability offers some comfort because you can use a design methodology that is sound and correctly create the design before attempting to program a device. This is traditionally how ASIC designs are carried out. Indeed, this is the way PLD designs will be executed in the future.

3.6 Multiple FPGA Simulations†

3.6.1 Introduction

When the gate count limit of a TI FPGA is reached the first step is to use a second FPGA in the design. You would then need the opportunity to make a complete timing simulation of several TI FPGAs using the TI design software. Each of the designs is first flattened to remove hierarchical designators. The various flat netlists are then treated as user soft macros for the simulator to process. In the following text of Section 3.6, you learn how to simulate multiple TI FPGAs.

3.6.2 Single Chip Design

The first step in creating a design is to set up the project directory using Viewfile. This helps organize the Workview data files in an easy way. For the FPGA $chip_1$, `\proj\chip1` was used and for FPGA $chip_n$, `\proj\chipn` was specified. Use Viewdraw to create the schematics for $chip_1$ up to $chip_n$. Normally, you would use I/O buffer macros with short net segments as I\O assignment for TI-ALS (with the net segments labeled). As shown in Figure 3-22 and Figure 3-23, for a multichip design you add the component IN to the net in front of INBUFs, and you add the component OUT to the net after OUTBUFs. The components IN and OUT are included in the built-in library of Workview. Both components are necessary to create a correct flat wire file later. Label the components IN and OUT with the same name as the connected nets.

You have now generated the complete designs for all FPGAs. After a functional simulation of each FPGA you are finished with the Viewlogic flow. The next step is to use TI-ALS to perform validation, I/O placement, place and route, delay extraction, and program export. Each design is now completely placed and routed, with the physical timing information back annotated to the Viewsim environment (`.vsm` file).

† Contributed by Peer Uhlemann, FPGA Applications, Texas Instruments Deutschland GmbH.

Figure 3-22. CHIP$_1$ Including I/O Buffers and Components IN/OUT

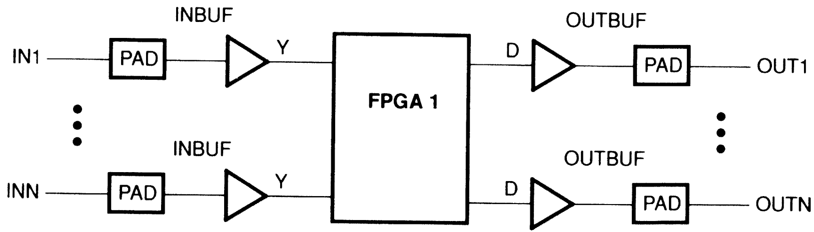
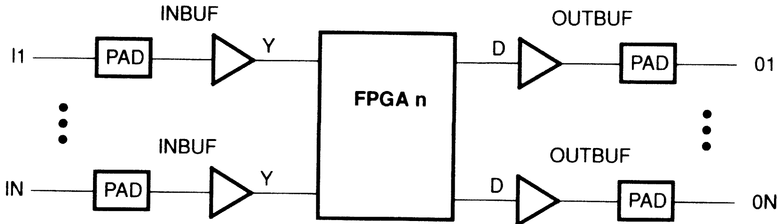


Figure 3-23. CHIP$_n$ Including I/O Buffers and Components IN/OUT



3.6.3 Multichip Design

1) Enter from DOS the following command to change to the right design directory.

   cd \proj\chip1

2) From the DOS window execute:

   vsm chip1 -w -d \designs\chip1\chip1.dtb

   This command causes the Viewsim wirelister, as shown in Figure 3-24, to create a flat wire file for the design. All physical delay information will be included in this wire file. The newly created chip$_1$ wire file will be stored under `\proj\chip1\chip1.1`

Figure 3-24. Viewsim Wirelister Usage

```
VIEWSIM WIRELISTER - V4.1.2; Workview 4.1.2 011792, 3000 Series
c Copyright 1985,1992 by Viewlogic Systems, Inc.
      Usage: vsm [proj] [level] [-ffile][-h|-s|-w|-t][-dtable]
        <proj>    Top level project.
        <level>    Level to wirelist.
        -f        Base file name for .vsm output.
```

```
-h      Generate LONG format wirelist (Default).
-s      Generate SHORT format wirelist.
-w      Generate WIR file output.
-t      Generate Viewfault format wirelist.
-d      Delay table file.
All arguments are optional.
```

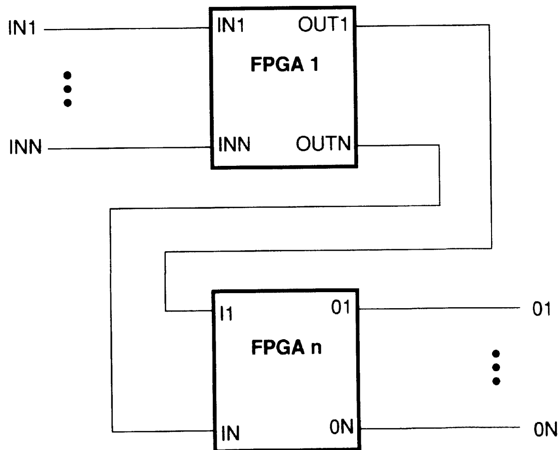
3) Repeat steps 1 and 2 for each design.

4) Use the Viewfile utility to create a new project `\proj\top`.

5) Create for each design a user-defined symbol. From the Viewdraw menus select | `Window` | `Open` | `Viewdraw` | `Symbol` | and enter the symbol name `CHIP1 ... CHIPn`. The pin label must be the same as the label of the components IN and OUT in the schematic `chip1 ... chipn`.

6) Capture the system-level schematics but do not use I/O buffers (Figure 3-25).

Figure 3-25. Schematic for Multichip Simulation



7) Exit Workview to edit the `\proj\top\viewdraw.ini` file. Use a pipe (|) to comment out the TI-FPGA libraries. This restriction will advise the Viewsim wirelister to use only the wire files to create a simulation netlist. Figure 3-26 shows the `viewdraw.ini` file for the current project `top`.

Figure 3-26. Modified `viewdraw.ini` *File*

```
| Viewdraw initialization file for Version 4.1
|
|
|
DIR [pw] D:\PROJ\TOP\
|DIR [rm] D:\ALS\LIB\A1000\CELLS (ACTELCELLS)
|DIR [rm] D:\ALS\LIB\A1000\MODELS (ACTELMODELS)
DIR [rm] D:\ALS\LIB\AUXLIB (BUILTIN)
```

8) Copy now the wire files from step 2 to the current project `top`. Type:

   `copy \proj\chip1\chip1.1 \proj\top\wir\`

   `copy \proj\chipn\chipn.1 \proj\top\wir\`

   The symbol and the wirelist are linked to each other by the names. That is the reason why the symbol name must be identical to the wirelist name.

9) Go back in the Viewdraw environment and read the design `top`. To simulate the whole design select `|Export | Wirelist | Viewsim |` and start Viewsim.

3.7 Using the TI-ALS Timer for Static Timing Analysis†

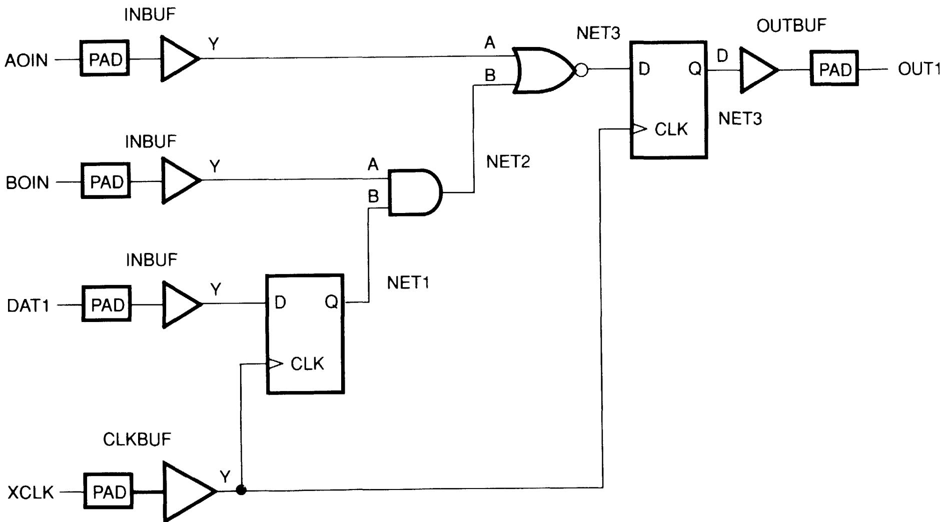
3.7.1 Introduction

The TI-ALS Timer is an interactive static timing analysis tool used to analyze path delays. In contrast to dynamic analysis, static analysis does not require vectors to display the timing of the design. Static timing analysis is useful for determining:

- Internal setup and hold timing checks
- Maximum operating frequency
- Input-to-output delays
- Clock skew
- External setup and hold requirements

Section 3.7 describes how the Timer can be used to analyze the timing associated with a design. The design selected for this purpose (shown in Figure 3-27) will allow you to look at the timings associated with both combinational and sequential logic.

Figure 3-27. Design Used for Timing Analysis



† Contributed by Mohan Maheswaran, Technical Marketing, Texas Instruments Ltd.

3.7.2 Entering the Timer and Saving Timer Results

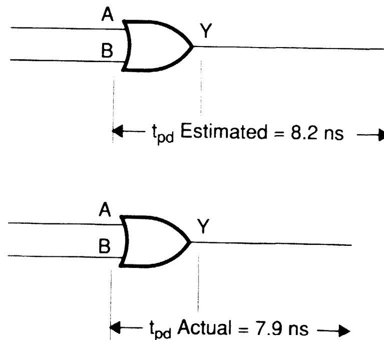
Prior to using the Timer the design must pass through the validation and configuration utilities in the TI-ALS system. The TI-ALS Timer reads the output files from the Validate and Config programs. These files are the Timer inputs. On entering the Timer the process and environment conditions must be set. The options allow the user to define worst-case or best-case process conditions and/or voltage/temperature conditions. The ranges of temperatures and voltages for commercial, industrial, and military parts, as represented in the Timer, are shown in the corresponding data sheets.

After TI-ALS is exited, the file called `/alsuser/user_name/user_name.log` is automatically written. This file contains all of the results shown on the screen during the use of the Timer and *must be copied to another file* if the results are not to be overwritten the next instance that TI-ALS is exited.

3.7.3 Prelayout Versus Postlayout

As shown in Figure 3-28, on entering the Timer you are given the choice of prelayout or postlayout timing. If prelayout timing is selected, statistical estimates are made for the interconnect delays. Postlayout delay simulations can be seen only after placing and routing the design. This mode is generally recommended for analysis. Prelayout mode can be useful for determining and assigning net criticality.

Figure 3-28. Prelayout Versus Postlayout Delays

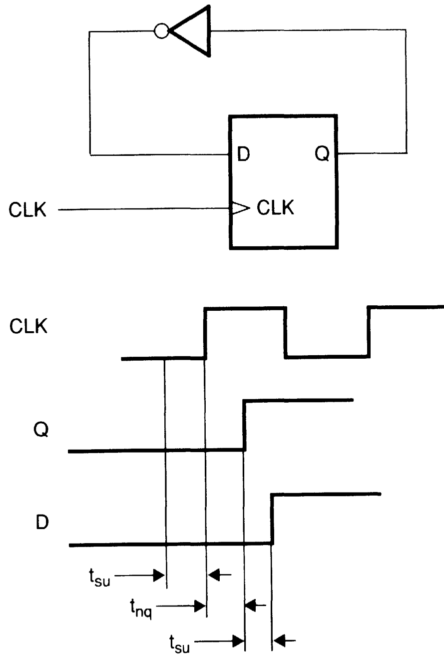


3.7.4 How the Timer Works

The TI-ALS Timer operates by determining path delays between specified start points and end points of the circuit. The group of start points is named

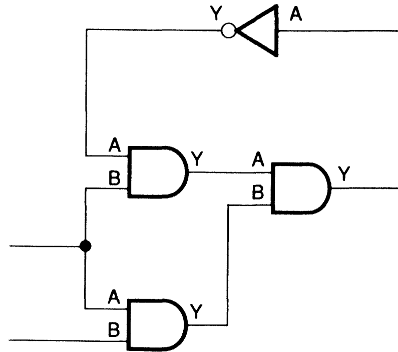
the `startset` and the group of end points is named the `endset`. The Timer has default sets of starting and end points. Two of the default sets are called `CLOCK` and `GATED`. These default sets define all the clock inputs as a set and all the gated outputs as a set. Thus, using the Timer you can calculate the worst-case register-to-register delay (Figure 3-29). In addition to setting the start and end points of any path, the Timer will rank the paths in order of delay magnitude and list them according to whether you request the longest or shortest delays.

Figure 3-29. Register-to-Register Delay



In practice the most commonly used *sets* will be the `startset` and `endset`, but there are also other sets available such as `breakset`, `tempset`, and `stopset`. These sets are used for defining points along an asynchronous path which needs to be broken for the Timer to analyze it. An asyncronous loop is a combinational loop that is not broken by a flip-flop stage. If the design contains asynchronous loops, the Timer cannot report the delay information for the loop path (Figure 3-30).

Figure 3-30. Asynchronous Loop



3.7.5 Creating and Modifying Sets

When the Timer is initialized, four predefined or default sets are created. They are defined as follows and in Figure 3-31.

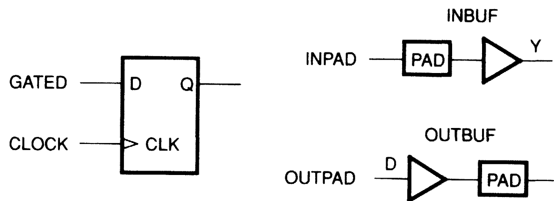
`CLOCK`—All sequential macro clock inputs

`GATED`—All sequential macro gated inputs

`INPAD`—All input buffer pads

`OUTPAD`—All output buffer pads

Figure 3-31. Four Default Sets



The **`set status`** command shows the current working sets. For this design example the current working sets are:

```
; Working startset 'CLOCK' contains 4 pins
; Working endset 'GATED' contains 2 pins
; Working stopset 'STOP' contains 0 pins
; Working passset 'PASS' contains 0 pins
```

This basically means that the start points are all the CLOCK pins (a pin is an input/output to a hard macro) and all the endpoints are GATED pins, which in this case are the D inputs to the flip-flops. To see the actual contents of each working set, the **set showset** command can be used, as shown in the following text.

Note:

Some macro pins represent more than one load. All flip-flops have some pins with two loads.

showset—Determines the content of a set.

showset clock

```
; Pin name          Net name        Macro
; FF2:CLK           CLK             DF1_1
; FF2:CLK           CLK             DF1_0
; FF1:CLK           CLK             DF1_1
; FF1:CLK           CLK             DF1_0
; 4 pins
```

showset gated

```
; Pin name          Net name        Macro
; FF1:D             NET3            DF1_0
; FF2:D             NET0            DF1_0
; 2 pins
```

showset inpad

```
; Pin name          Net name        Macro
; DAT1:PAD          DAT1            INBUF
; A0IN:PAD          A0IN            INBUF
; B0IN:PAD          B0IN            INBUF
; XCLK:PAD          XCLK            CLKBUF_0
; 4 pins
```

showset outpad

```
; Pin name          Net name        Macro
; $1I5:PAD          OUT1            OUTBUF
; 1 pins
```

3.7.6 Changing Sets

The Timer analysis applies to the currently defined working set. To change the current starting set or ending set simply use the commands:

`startset` ***`set_name`***

`endset` ***`set_name`***

`startset clock`—sets starting point to clock

`endset gated`—sets ending point to gate

`startset inpad`—sets starting point to input pad

`endset outpad`—sets ending point to output pad

where *`set_name`* is the name of either a previously defined set or a new set. After you have defined a starting and ending set you can add or delete starting points from the set using the following commands:

`addstart` ***`instance_name`*** **`:pin`**

`remstart` ***`instance_name`*** **`:pin`**

`addend` ***`instance_name`*** **`:pin`**

`remend` ***`instance_name`*** **`:pin`**

3.7.7 Maximum Operating Frequency

In a synchronous design, the maximum operating frequency for that design is defined as the inverse of the longest register-to-register delay. The register-to-register delay is the delay from the clock of a flip-flop to the data input of a flip-flop, including all combinational gate delays between the registers and the required setup time on the ending register.

In the design example there are two flip-flops. If you use the default sets of `startset clock` and `endset gated` then the longest delay is as follows:

```
;  1st  longest path to all endpins

; Rank Total Start pin        First Net     End Net       End pin

;   0   31.6 FF1:CLK           NET1          NET3           FF2:D
```

Using the Expand utility of the Timer, you can expand the path ranked 0 to obtain the following breakdown showing all the delays associated with the path from the clock input of flip-flop1 to the D input of flip-flop2.

```
;  1st longest path to $FF2:D (falling) (Rank: 0)

; Total Delay Typ Load Macro      Start pin        Net name
```

```
;  31.8    6.7 Tsu   0  DF1        $FF2:D
;  25.1    6.9 Tpd   1  NOR2       $1I4:B                 NET3
;  18.2    7.4 Tpd   1  AND2       $1I26:B                NET2
;  10.8   10.8 Tcq   2  DF1        $FF1:CLK               NET1
;  0.0     0.0 Psk   4             $FF2:CLK               CLK
```

The longest register-to-register delay is 31.8 ns giving a maximum operating frequency of 31.5 MHz. The total delay is accumulated by combining the clock skew, the clock-to-Q of flip-flop1, the gate propagation delays and the setup time to flip-flop2. The delays are most significant for those macros that have more than one level of logic. At this point it is prudent to mention that this maximum frequency of operation is conservative because of the following key points:

- The actual delays can be heavily influenced by the manual assignment of pins and macros as well as the assignment of criticality.
- The use of an advanced logic enhancer and synthesizer tool can optimize the design for both area and speed.
- The Timer has been set to use the standard commercial FPGA and not the fastest speed-graded FPGA.

3.7.8 Input-to-Output Delay

While the register-to-register delays produce the maximum operating frequency for the design internally within the chip, you can also use the Timer to give the input-to-output delays of the chip. Input-to-output delay is typically used for determining the delay on combinational paths from input pad to output pad. This is done by setting the startset to INPAD and the endset to OUTPAD. Then pick up the longest or shortest delays, such as the clock-to-gated, and expand on these delays.

```
;  1st  longest path to all endpins
; Rank Total Start pin        First Net    End Net      End pin
;  0   26.4 XCLK:PAD          CLK          OUT1         $1I5:PAD
; 1 pins
;  1st longest path to $1I5:PAD (falling) (Rank: 0)
; Total Delay Typ Load Macro      Start pin         Net name
;  26.4   9.8 Tpd   0  OUTBUF     $1I5:D                OUT1
```

```
;  16.6    5.7 Tpd   2  DF1        $1I3:CLK        NET4
;  10.9   10.9 Tpd   4  CLKBUF     XCLK:PAD        CLK
```

Thus the 26.4-ns input-to-output delay consists of the propagation delays through the clock buffer and output buffer and the delay through the flip-flop.

3.7.9 Setup and Hold Time Requirements

Internal setup time checking is done implicitly. For a synchronous design, the maximum period displayed by using a longest command will include the required setup time for a sequential macro. This setup time is essentially the time interval, for which the data must be present, immediately preceding the clock transition in order for the data to be recognized. See that the setup time for flip-flop2 is 6.7 ns.

```
;  1st longest path to $1I3:D (falling) (Rank: 0)
; Total Delay Typ Load Macro      Start pin       Net name
;  31.8   6.7 Tsu   0  DF1        FF2:D
;  25.1   6.9 Tpd   1  NOR2       $1I4:B          NET3
;  18.2   7.4 Tpd   1  AND2       $1I26:B         NET2
;  10.8  10.8 Tcq   2  DF1        FF1:CLK         NET1
;   0.0   0.0 Psk   4             FF2:CLK         CLK
```

If the **shortest** command is used then the Timer will check internal hold time violations. The hold time is essentially the time interval immediately following the clock transition, during which the data input must be present in order for the input to be recognized. In this case, the hold time for flip-flop D is 0 ns.

```
;  1st shortest path to $1I3:D (rising) (Rank: 0)
; Total Delay Typ Load Macro      Start pin       Net name
;  24.5   0.0 Thd   0  DF1        FF2:D
;  24.5   7.2 Tpd   1  NOR2       $1I4:B          NET3
;  17.3   6.9 Tpd   1  AND2       $1I26:B         NET2
;  10.4  10.4 Tcq   2  DF1        FF1:CLK         NET1
;   0.0   0.0 Psk   4             FF2:CLK         CLK
```

3.7.10 External Setup Time Requirements

External setup time for the device is a critical timing parameter in a synchronous system design. The timing relationship between the external system clock and the external data must ensure that the internal setup time requirement of the sequential elements is met. Calculating the external setup time requirement is a two step process. First, determine the clock pad to register clock input delay. The external setup time required must satisfy the following equation:

$$\text{Clock path delay} - \text{Data path delay} + t_{su}\ \text{ext} > t_{su}\ \text{int}$$

where t_{su} int is the internal setup time requirement for a sequential macro and t_{su} ext is the external setup time requirement for the data that must be provided by the system for proper device operation.

3.7.11 Clock Skew

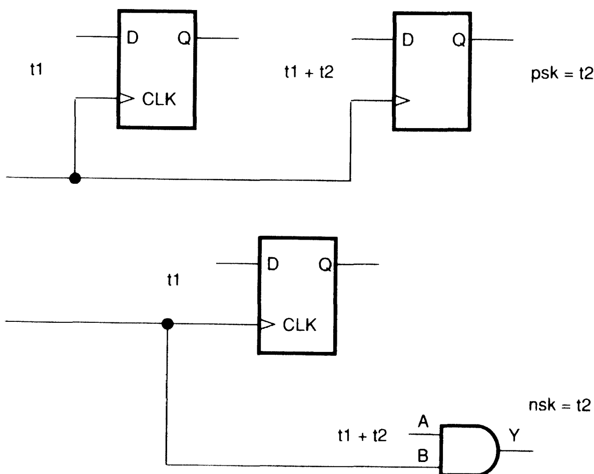
Clock skew is the delay variation on the clock net connecting two or more sequential elements, and is the difference between the largest delay and the smallest delay. Clock skew will depend on the number of loads on the clock circuit and the degree of clock balancing used in place and route. The clock balancing utility allows you to influence the placement and routing of clocked macros to minimize skew, if required, by balancing loads on the clock distribution network. Flip-flops located on the same row will have negligible skew, but flip-flops located on different rows will have skew dependent on the difference in the number of clock loads on the rows. Flip-flops that are clocked by I/Os will have skew relative to flip-flops clocked by the clock driver.

The `Skewmode` option in the Timer determines whether or not pin-clock skew time is calculated. Pin-clock skew (P_{sk}) is a measure of the difference in delays for the clock line to reach more than one flip-flop when there is a different delay path between flip-flops.

Net skew (N_{sk}) is calculated when flip-flops on a common clock drive a nonstorage macro (such as a gate). Net skew is the difference between the clock net delay, or the path being examined, and the smallest clock delay or the common clock network.

When timing constraints are significant, the clock skew should be considered along with the rest of the delays in the circuit. See Figure 3-32. If there is a problem with clock skew, use the load or clock balancing features.

Figure 3-32. Clock Skew



3.7.12 Pins Selection

The Timer sums delays along a path, which has a single starting and ending pin. Both the starting and ending point are macro inputs. For I/O buffers, the pad is the starting point for INBUFS and the ending point for OUTBUFs and TRIBUFs. The pad of a BIBUF can be either a starting point, ending point or both. By selecting the starting and ending points, the Timer will calculate the delay.

3.7.13 Other Timer Options and Utilities

The following discussions are based on a synchronous design that has no user-defined macros but there are also Timer utilities to aid in analyzing timings associated with asyncronous circuits and user-defined macros.

3.7.13.1 Names and Level

The command **names** specifies the name of a pin from a higher level in the design hierarchy than the level found in the TPC10 series macro library, making pins easier to reference. When **names** is selected `ON` you do not need to refer to the pin's full pathname. Only the user-created macro name and its corresponding pin are needed. When **names** is `OFF`, you must reference pins down to the TPC10 macro library level. The `Level` option will allow you to set one of three options: `Hard`, `Soft`, or `User`. The Timer will display the delay information for the entire soft or user-defined macro if `Soft`

or `User` is selected or break down the internal macro paths to show the internal delays at hard-macro level.

3.7.13.2 Other Set Commands

- **`stopset`**

  When an asynchronous loop is detected, the Timer issues a warning. Asynchronous loops must be broken for the Timer to analyze the delays. This is facilitated in the Timer by using a **`stopset`** command. By creating a stopset containing a point in an asynchronous loop, Timer is able to complete the analysis for the circuit.

- **`breakset`**

  When asynchronous loops are detected, a special set is generated that contains a possible breakpoint. This set is called `breaki`, where `i` is an integer. If three asynchronous loops are detected, then the break sets are created automatically (`break1`, etc.), one for each loop. The easiest way to add these break sets to the stopset is to use the **`orstop`** command.

  ```
  > orstop break1
  ```

  All of the required break points are then merged to the stopset, and any of the analysis commands can be executed.

- **`tempset`**

  In addition to break sets, Timer also generates temp sets (`temp1`, etc.), one temp set for each asynchronous loop. If you want to chose your own break points, you can display the contents of the **`tempset`** by using the **`showset`** command to help determine which point to use and then use the **`addstop`** command to add the points to the stop set.

The TI-ALS Timer is a comprehensive timing analysis tool which offers the ability to carry out a complete static timing analysis of both sequential and combinational circuits. It should be used in conjunction with a simulator to further enhance the design flow. With careful digital design practices all the necessary timing information can be extracted for a design.

3.8 Critical Path Analysis for FPGAs†

Device speed, or timing, is a critical aspect of system design. A realistic estimate of the achievable system speed is often required early in the design phase to avoid waste of valuable design time. System speed, of course, depends on the operation of all system components. This section describes how to estimate the timing constraints of a field-programmable gate array (FPGA) design. Remember, the FPGA is just one component within the design—other devices also affect the system speed.

3.8.1 Methodology

Three things are needed to estimate system speed for a TPC10 series FPGA design:

- *TPC10 Series Family Data Sheet* available from Texas Instruments.
- *TPC10 Series Data Sheet Supplement* available from Texas Instruments. The *Pin Loading* section of this supplement is contained in Chapter 2 of the FPGA Data Manual.
- Knowledge of the design's critical path (The critical path is entirely design-dependent).

Section 3.8 begins by defining terms associated with estimating system speed (critical path, logic levels, and fan-out). It then provides examples of estimating the propagation delay of a combinational path and the system speed of a sequential path.

When following the procedures outlined in this section, remember that all data sheet delays are typical delays. To determine timing delays for a particular application, always consider the voltage and temperature conditions and derate the typical data sheet values accordingly.

3.8.1.1 Critical Path

What is a critical path? It is a path in the design which must meet certain critical timing requirements in order for the system to function properly. A critical path can be composed of any combination of hard or soft macros, and input and output buffers for either combinational or sequential logic paths. This section addresses each element of the critical path separately.

3.8.1.2 Logic Levels

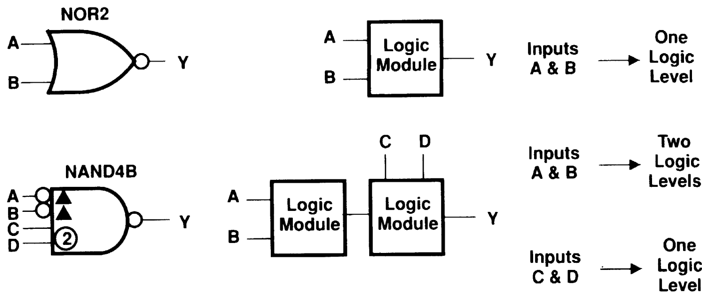
After the critical path has been identified, the next step is to determine the hard or soft macros that will be used to implement the critical path from the

† Contributed by Anne Phillips, Dan Powers, and Mary Werling, FPGA Applications, Texas Instruments Incorporated.

TPC10 Series Family Data Sheet and to note the respective logic levels for each of those hard or soft macros. The terms *levels of logic* or *logic levels* refers to the number of logic modules that an input must incorporate to perform the desired logic function and reach the output.

Hard macros have either one or two logic levels. Two levels of logic means that two logic modules are required to implement this function in the TI FPGA architecture. The filled triangle symbol on the inputs of hard macros denotes two levels of logic. Hard macros without the triangle have only one level of logic. Figure 3-33 shows an example of one and two-level hard macros. For a two-level hard macro, the logic module interconnection is always consistent and the timing characteristics remain unchanged.

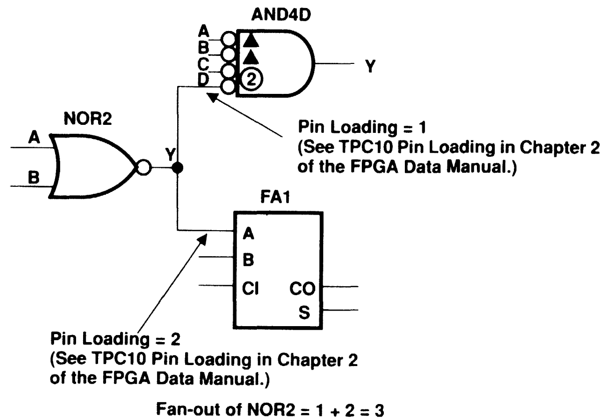
Figure 3-33. Logic Levels



A soft macro is a collection of hard macros configured to make a more complex function. Logic levels for soft macros are listed in the data sheet. Soft macros can have many logic levels (built using many logic modules).

Fan-out is the sum of the pin loads for all macro inputs connected to the output of a particular macro. The pin loading for every hard and soft macro is listed in the data sheet. Figure 3-34 illustrates the fan-out of an element in a design.

Figure 3-34. Fan-out



3.8.2 Example 1—Combinational Path

3.8.2.1 Delay Caused by a Single-Level Hard Macro

With the help of the *TPC10 Series Family Data Sheet*, typical delays caused by hard macros can be easily determined. For this example, the NOR2 hard macro is used.

Levels of Logic:

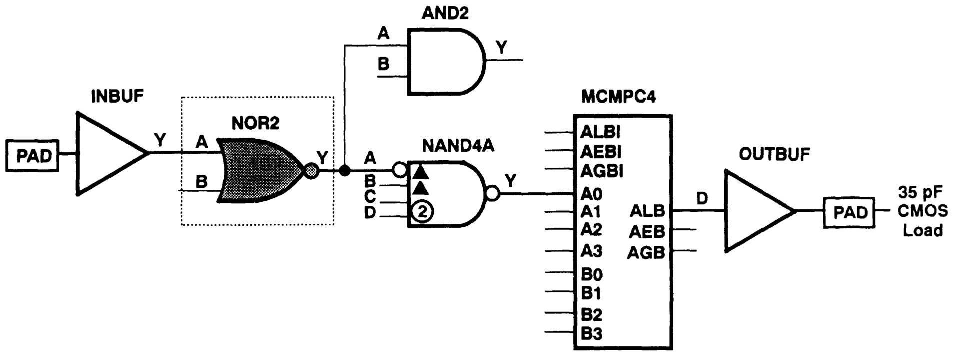
As a first step, you determine the number of logic levels for NOR2, the hard macro. The critical path is through the A input of NOR2. By looking at the NOR2 symbol in the data sheet you can see that there is no triangle on the A input. This indicates that logic levels = 1.

NOR2 Fan-out:

Next, the fan-out for NOR2 is determined. In this design (Figure 3-35), the output of NOR2 feeds the A input of NAND4A and the A input of AND2. The *TPC10 Series Data Sheet* reveals that the A input of the NAND4A macro has a pin loading of 1 and that the A input of AND2 also has a pin loading of 1.

Simply adding the pin loading of all inputs tied to the output of NOR2, or 1 + 1, provides the fan-out. In this case, the fan-out = 2.

Figure 3-35. Single-Level Hard Macro Delay



The *TPC10 Series Family Data Sheet* contains a table with information on single-level logic module hard-macro switching characteristics. This information is needed to determine the typical delay caused by NOR2. According to that table, a hard macro in a critical path with one level of logic and a fan-out of 2 has a delay of 5.8 ns. This value includes a statistical delay estimate for the routing between the NOR2 output and the NAND4A input.

Note:

Many FPGA vendors do not include delays due to routing in their data sheets.

3.8.2.2 Delay Caused by a Double-Level Hard Macro

Levels of Logic:

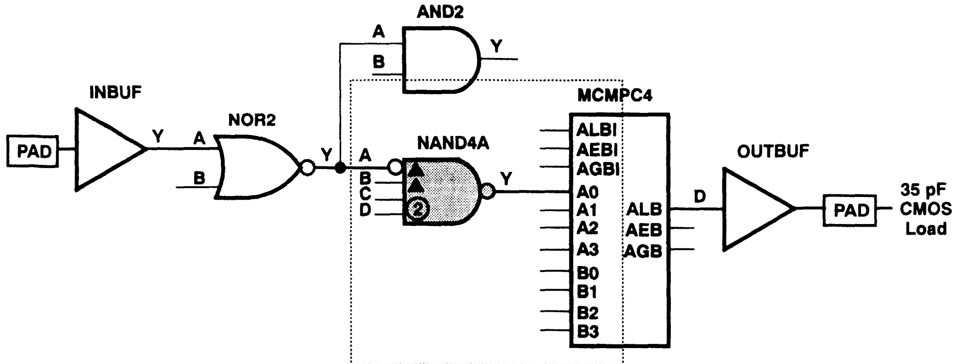
As shown in Figure 3-36, the critical path is through the A input of the NAND4A hard macro. By looking in the data sheet, you can see that there is a triangle on the A input of the NAND4A. This denotes that there are two levels of logic through the A input of this hard macro.

NAND4A Fan-out:

Next, the fan-out for NAND4A is determined. In this design (Figure 3-36), the output of NAND4A feeds the A0 input of MCMPC4. The TPC10 pin loading information reveals that the A0 input of MCMPC4 has a pin loading of 3. Therefore, the fan-out of the NAND4A is 3.

The data sheet contains a table with information on double-level logic module hard macro switching characteristics. It shows that a hard macro with two levels of logic and a fan-out of 3 has a delay of 10 ns.

Figure 3-36. Double-Level Hard Macro Delay



3.8.2.3 Delay Caused by a Soft Macro

Levels of Logic:

The levels of logic for a soft macro are again given in the data sheet. The MCMPC4 soft macro (shown in Figure 3-37) has four logic levels.

Fan-out:

It is impossible to determine the load on each of the hard macros within a soft macro without examining the soft macro schematic. Since the soft macro schematics are available only in the CAE environment, the load on the internal hard macros will have to be estimated.

1) Calculate the delay of n – 1 levels of logic, where n is the total number of logic levels in the soft macro. A conservative estimate of three loads should be used to calculate the delay of n – 1 levels of logic. The delay for n – 1 levels is:

   delay = (n – 1) * 6.2

   where 6.2 ns is the delay for a single-level hard macro using critical routing and a fan-out of 3.

2) Determine the load on the output of the macro. The delay on the nth level of logic is simply the delay of a single-level hard macro using critical routing and having the appropriate fan-out.

3) Add the values from steps 1 and 2 to obtain the total delay of the soft macro.

The *TPC10 Series Data Sheet* shows that the soft macro used in this example (Figure 3-37) has (n) = 4 logic levels.

1) The calculation for the delay of the first three levels of logic (n – 1 levels) is:

   (n – 1) * 6.2 = (4 – 1) * 6.2 = 18.6 ns

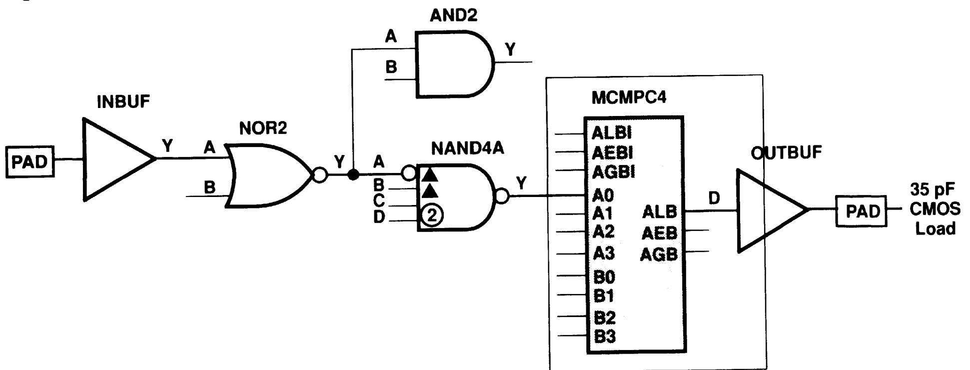
2) Determine the fan-out for the fourth logic level in the MCMPC4 soft macro by looking at the TPC10 pin loading information to find the pin loading on the D input of the OUTBUF macro. The fan-out is 1.

   Determine the delay for the fourth level of logic by looking at the *TPC10 Series Family Data Sheet*. It shows the single-level logic module hard macro switching characteristics for a critical path with a fan-out of 1. The delay for the fourth logic level is 5.4 ns.

3) Find the total delay by adding the delay of the first three levels, 18.6 ns, to the delay of the fourth level, 5.4 ns. Thus, the total delay in the MCMPC4 soft macro is:

   Total delay = 18.6 ns + 5.4 ns = 24 ns

Figure 3-37. Soft Macro Delay



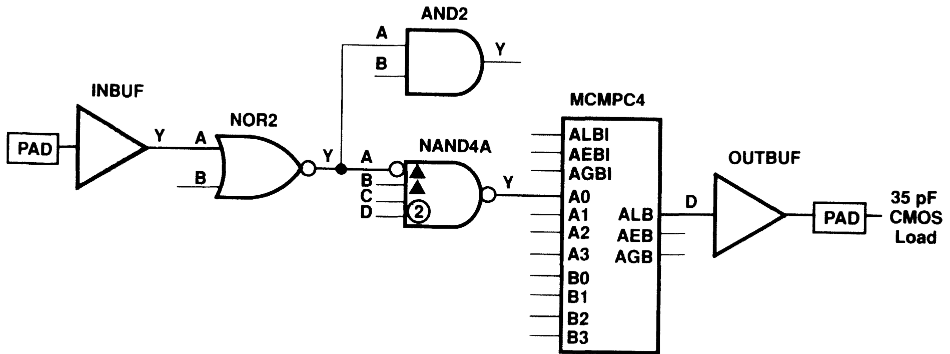
3.8.2.4 Delay Caused by an Input Buffer

Input buffer delays (and bidirectional input buffers) are calculated much like hard macro delays (Figure 3-38). The only difference is that the propagation delay for the high-to-low transition is always used for critical paths.

High-to-low transition delays are chosen because they are always greater than low-to-high transition delays, as specified in the data sheet.

The data sheet lists the delay for an input buffer with a fan-out of 1 as 6.9 ns.

Figure 3-38. Input Buffer Delay



3.8.2.5 Delay Caused by an Output Buffer

Output buffer delays (and three-state and bidirectional output buffers) are specified in the *TPC10 Series Family Data Sheet* for driving CMOS or TTL loads.

The output buffer delays are characterized for a load capacitance of 50 pF. To calculate a propagation delay for an output buffer with a load other than 50 pF, the delta t_{pd} factors from the data sheet must be used.

The delay specified for the critical path evaluation should be the greater of the low-to-high propagation delay, or the high-to-low propagation delay.

The following equation calculates output buffer delay with a load capacitance other than 50 pF.

$$t_{pd}(\text{actual}) = t_{pd}(50\text{ pF}) + [\text{delta } t_{pd} * (C(\text{actual}) - 50)]$$

The delay caused by the output buffer shown in Figure 3-39 is determined with the help of further information from the *TPC10 Series Family Data Sheet.*

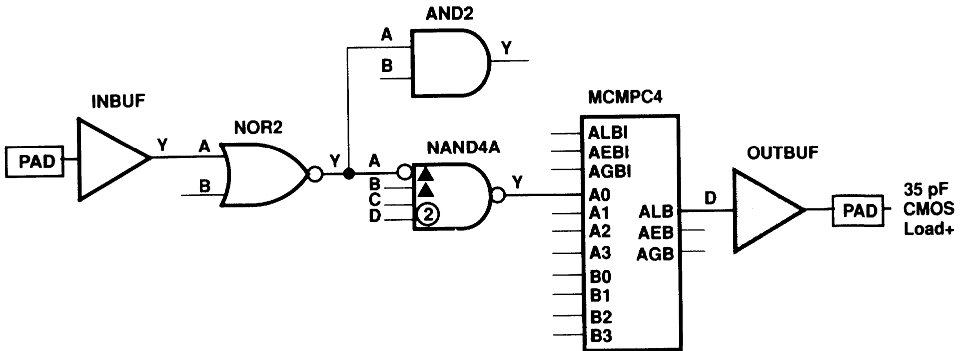
$$t_{PHL} = (3.9) + [.03 * (35 - 50)] = 3.5\text{ ns}$$

$$t_{PHL} = (7.2) + [.07 * (35 - 50)] = 6.2\text{ ns}$$

Output buffer delay = 6.2 ns

As you can see, the example CMOS load used here is 35 pF.

Figure 3-39. Output Buffer Delay



3.8.2.6 *Combinational Critical Path*

The components of the critical path have been identified and the delays have been calculated. The next step is to determine the cumulative timing delays of all elements in the critical path.

The critical path delay for a combinational path is the sum of all delay elements in the path. Note that the delay caused by the AND2 macro has not been added because AND2 is not in the critical path.

INBUF+NOR2+NAND4A+MCMPC4+OUTBUF = 6.9 + 5.8 + 10 + 24 + 6.2

Total delay = 52.9 ns

The total estimated delay is a typical value. To account for temperature and voltage factors the typical delay is derated with the derating factors shown in the *TPC10 Series Family Data Sheet*. For example, to obtain the estimated worst-case delay for a design under commercial operating conditions the typical delay value is multiplied by 1.54.

The following calculation is for the example of a combinational path:

Worst-case delay = 52.9 x 1.54 = 81.5 ns

3.8.3 Example 2—Sequential Critical Path

The technique for analyzing a sequential critical path differs from the technique used for combinational paths. Setup and hold characteristics have to be considered and the critical path has to be broken into segments.

Hold time is the time for which a signal must be held at a specified state after the clock transition.

Setup time (t_{su}) is the time for which a signal must be maintained at a constant valid state before the clock transition. Setup times for both flip-flops and latches are given in the data sheet.

Propagation delay (t_{pd}) of flip-flops and latches also must be taken into consideration when evaluating the achievable system speed of a sequential path. Propagation delays for flip-flops and latches are also given in the data sheet.

Sequential paths must be broken into segments for critical path analysis:

- ❑ Inputs to clocked macros
- ❑ Clocked macros to clocked macros
- ❑ Clocked macros to outputs

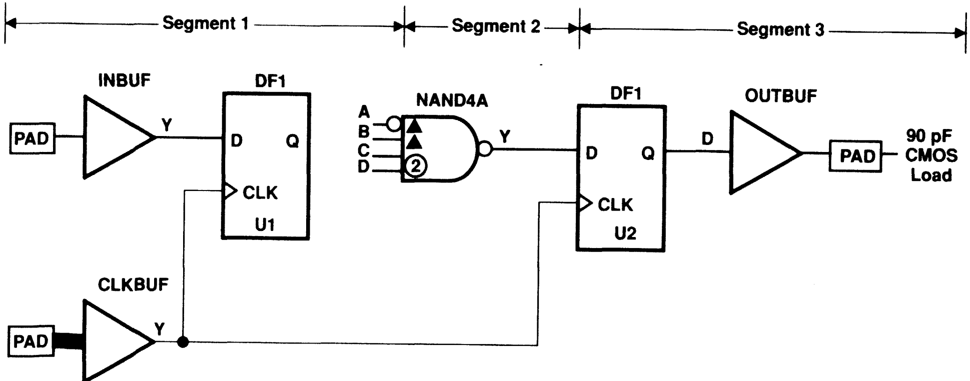
The example in Figure 3-40 is divided into three segments. The first segment is between the input buffer and the first clocked macro. The second segment is from one clocked macro to another. The third segment is from the last clocked macro to the output buffer. The segments are compared and the one with the greatest delay determines the achievable system speed for the device.

With information in the *TPC10 Series Family Data Sheet*, delays for the three segments are quite easily determined.

Delay caused by segment 1:

$$t_{seg}\,1 = t_{pd}(\text{inbuf}) + t_{su}(\text{U1}) = 6.9 + 3.9 = 10.8\text{ ns}$$

Figure 3-40. Sequential Path Delay



Delay caused by segment 2:

$$t_{seg}\,2 = t_{pd}(U1) + t_{pd}(NAND4A) + t_{su}(U2) = 5.4 + 9.2 + 3.9 = 18.5 \text{ ns}$$

Delay caused by segment 3:

$$t_{seg}\,3 = t_{pd}(U2) + t_{pd}(outbuf) = 5.4 + 10 = 15.4 \text{ ns}$$

In the example shown in Figure 3-40, segment 2 has the greatest path delay, 18.5 ns, so it determines the critical path delay. Again, this is a typical value. To obtain the estimated worst-case delay, multiply by 1.54.

Worst-case delay value = 18.5 x 1.54 = 28.5 ns

System clock frequency is calculated with the following simple equation:

frequency = 1/delay

frequency = 1/28.5 ns

frequency = 35 MHz

In this case, the clock buffer delay has been completely neglected. This is a reasonable assumption for many applications. Generally, the clock buffer delay becomes a consideration only if its delay exceeds the largest segment delay. This is possible in register or latch-intensive designs where the clock network is heavily loaded. In such situations, the clock buffer delay could be the limiting factor.

Section 3.8 has described how to estimate the device speed of a TI TPC10 series FPGA by calculating the delay caused by the various elements in the design's critical path. Examples of sequential and combinational critical paths are used to show how the calculations are performed. The result of the calculations provides an estimate of the speed with which the FPGA is likely to perform. Similar calculations can be made for TPC12 series FPGAs.

3.8.3.1 Alternate Method—10-ns Rule

A quick method to estimate achievable system speed is to assume a 10-ns delay for each hard macro and 10 ns/logic level for each soft macro in the path.

3.8.3.2 Preferred Method

The most accurate way to estimate achievable system speed for a TI FPGA is to capture the critical path and analyze it using the TI Action Logic System.

3.9 Understanding How TI-ALS Places and Routes[†]

3.9.1 Introduction

The TI FPGAs are based on channeled array architecture consisting of an array of logic modules and interconnect channels that contain horizontal metal lines. There are also metal tracks running vertically over the logic modules. Each horizontal-vertical track intersection contains an antifuse which can be programmed to form a low resistance path between the intersecting metal tracks.

Among the vertical tracks there are long vertical tracks (LVT) and module output tracks. Section 3.9 shows how you can interpret the files which are generated by the TI-ALS during the automatic place and route (APR) to get a better understanding about the array architecture and how the routing tracks are used by the APR.

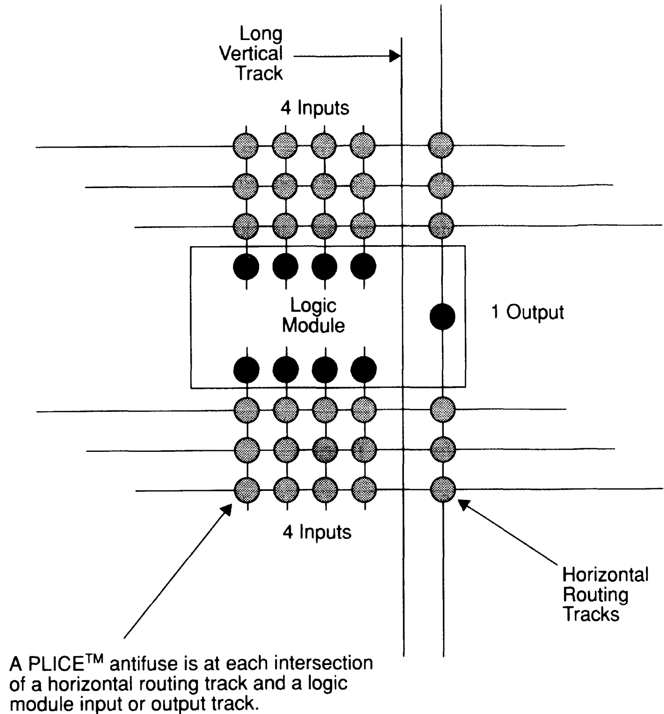
3.9.2 Logic Module Vertical Tracks

Each TPC10 series logic module has eight inputs and one output as shown in Figure 3-41. The eight inputs are arranged so that four inputs are available to the routing channel above the module, and four inputs are available to the channel below the module. The module output is available to four routing channels, two channels above the module and two channels below the module.

Figure 3-41 shows also one long vertical track. While the module output track is dedicated to one module, the LVT can be used by any module.

[†] Contributed by Dung Tu, Technical Marketing, Texas Instruments Deutschland GmbH.

Figure 3-41. Logic Module with Output Track and Long Vertical Track



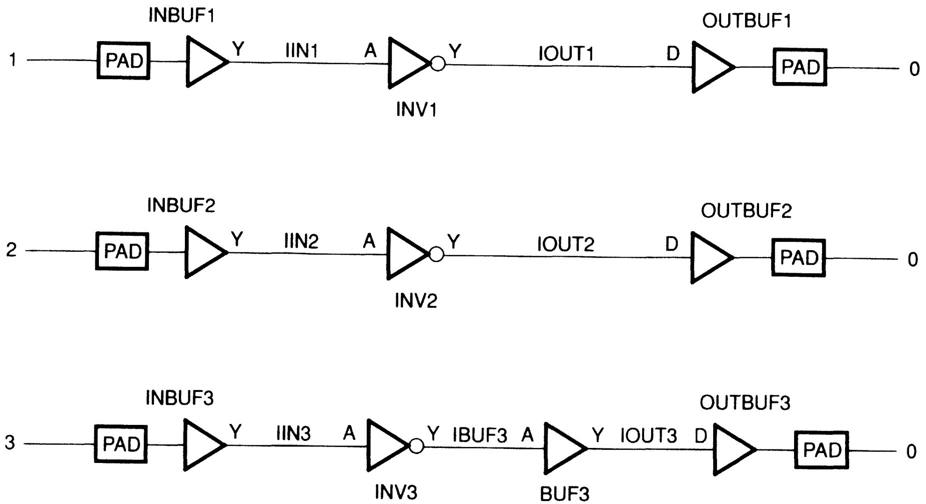
3.9.3 Example Circuit

Figure 3-42 shows the example circuit—called `LONGLINE`—which is used to investigate how APR works. The circuit does nothing useful. The basic idea is to look at a signal path which consists of an input buffer at one end, an output buffer at the other end, and anything in between, which is built with one logic module (such as a simple inverter). This signal path is implemented in several different ways, using manual pin assignment, to discover how the APR does the placement and routing. In this case a TPC1010 was used in a 68-pin PLCC package.

In the first case, path 1, manual pin assignment is used to force the input pin to the top side of the array (pin 9) and the output pin to the bottom side (pin 27).

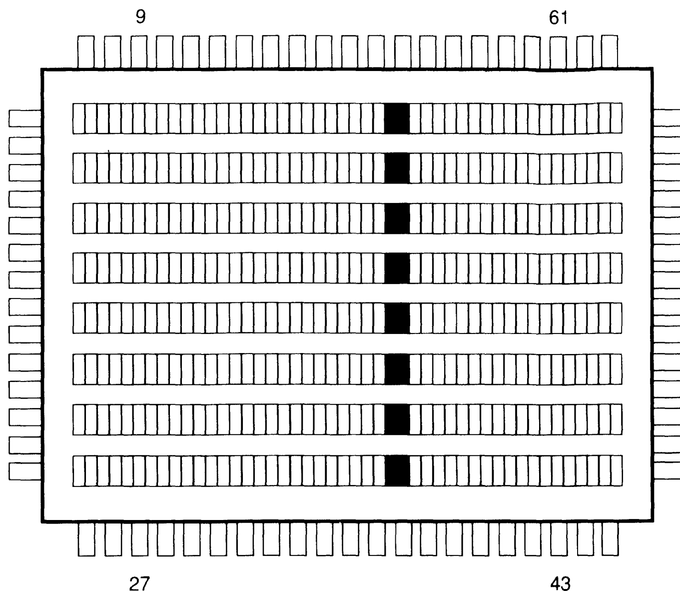
For the second case, path 2, the APR does the automatic pin assignment.

Figure 3-42. Evaluation Circuit



In the third case, path 3, the input is again forced to the top side (pin 61) and the output to the bottom side (pin 43) but a buffer is added between the inverter and the output buffer. Figure 3-43 shows the pinout of a TPC1010 in a 68-pin package.

Figure 3-43. TPC1010 Pinout, 68-Pin Package



3.9.4 Pin Assignment, Pin File

During place and route, the TI-ALS generates several files which are important for understanding how the APR works.

There are two files which show the locations of the pins. The `.ipf` file shows only the pins which are fixed by the manual pin edit. The `.pin` file shows the locations of all pins.

Figure 3-44. Example `.ipf` File Shows the Fixed Pins

```
DEF LONGLINE.

NET IN3; ; PIN:61.

USE ; INBUF3; FIX.

NET OUT3; ; PIN:43.

USE ; OUTBUF3; FIX.

NET IN1; ; PIN:9.
```

```
USE ; INBUF1; FIX.

NET OUT1; ; PIN:27.

USE ; OUTBUF1; FIX.

END.
```

Figure 3-45. Example `.pin` *File Shows All Fixed and Unfixed Pins*

```
DEF LONGLINE.

NET IN2;;

  PIN:67.        <--- unfixed, assigned by APR

NET IN1;;

  PIN:9,

  FIX:''.

NET OUT2;;

  PIN:68.        <--- unfixed, assigned by APR

NET IN3;;

  PIN:61,

  FIX:''.

NET OUT1;;

  PIN:27,

  FIX:''.

NET OUT3;;

  PIN:43,

  FIX:''.

END.
```

Looking at the `.ipf` file (Figure 3-44) you can see that for path 1 and 3 the input and output are fixed at the pins on the top and the bottom of the array.

The `.pin` file (Figure 3-45) shows that, for path 2, TI-ALS has used pin 67 as input and pin 68 as output—two pins which are adjacent to each other.

3.9.5 Location File, Placement File

Figure 3-46 illustrates the location file (`.loc`) of the design. The location file shows where the APR has placed the components. For example, the input buffer INBUF1 is placed in a logic module which is at column number 6, row number 6, and the location is user-specified.

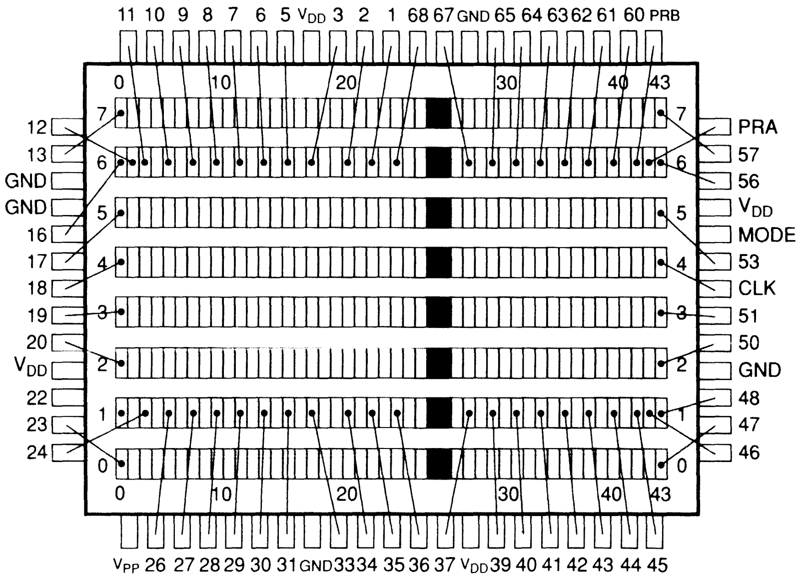
Figure 3-46. Location (`.loc`) File Shows the Component Locations

```
DEF LONGLINE.

USE ; INBUF2;

  XY:27%6.

USE ; INBUF1;       <--- Component label

  XY:6%6,           <--- XY coordinate: Column # % Row #

  FIX:''.           <--- Fixed by manual Pin Edit

USE ; OUTBUF2;

  XY:23%6.

USE ; INBUF3;

  XY:37%6,

  FIX:''.

USE ; OUTBUF1;

  XY:6%1,

  FIX:''.

USE ; OUTBUF3;

  XY:37%1,

  FIX:''.

USE ; BUF3;

  XY:37%3.

USE ; INV2;

  XY:27%7.

USE ; INV1;
```

```
  XY:6%0.
USE ; INV3;
  XY:37%5.
END.
```

Figure 3-47 shows the XY coordinates and the floorplan for the TPC1010. This device consists of eight rows and 44 columns of logic modules. The logic module in the lower left corner is labeled (X = 0, Y = 0) and the top right corner is labeled (X = 43, Y = 7).

Figure 3-47. TPC1010 Floorplan



There are a total of nine routing channels. The bottom routing channel (below row 0) is labeled channel 0, the top channel (above row 7) is labeled channel 8.

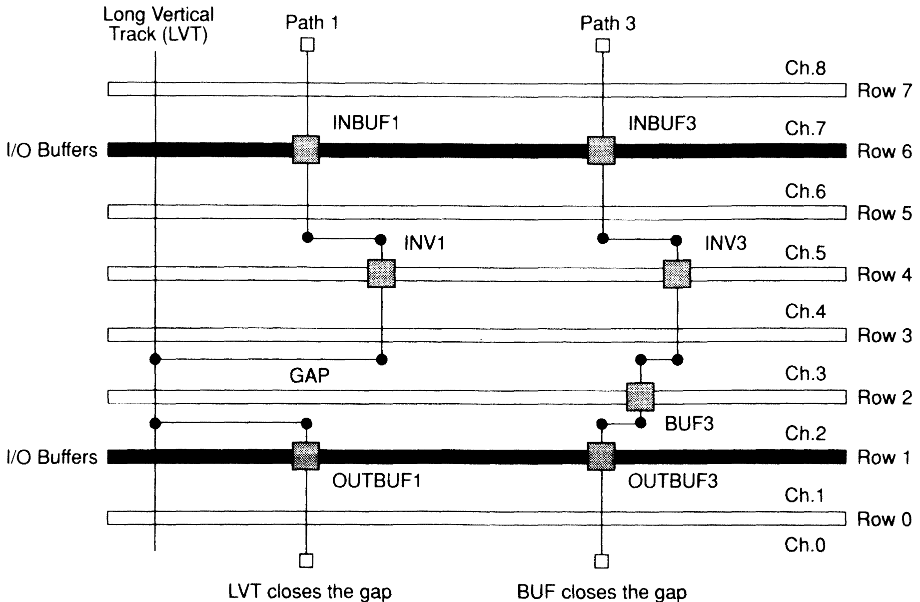
The APR also generates a placement information file with the extension `.pli` that informs which nets were routed using long vertical tracks (Figure 3-48).

Figure 3-48. Placement Information (`.pli`) File

```
One net of the design, net IIN1 of the signal path 1, needs a
LVT for routing. This net is driven at location (X = 6, Y=6) by
INBUF1. The LVT runs over column 6 and spans from channel 8 to
channel 0.
Average horizontal net length is 1.71429 columns.
No nets need long horizontal tracks.
A total of one net is needed long vertical tracks.
Automatic placement completed successfully.

The design has 7 routed nets (excluding GND, VCC, global CLKs)
. 6 nets need no long vertical tracks, av. horiz. length for
these nets is 1.6667, av. fanout 1.0000.

1 nets needs to use long vertical routing tracks. 1 of such
tracks are of the standard type (LVT), av. horiz. length for
these nets is 2.0000, av. fanout 1.0000.

List of all nets using standard long vertical tracks (LVT) to
route:
The net IIN1 driven at location XY = (6, 6) uses an LVT.
LVT data: column = 6, Y-span = (8, 0).
Net data: fanout = 1, Y-spread of inputs = (0, 1).
```

Figure 3-49 depicts why a LVT is needed for path 1.

Figure 3-49. Long Vertical Track



The INBUF1 is fixed at row 6, the OUTBUF1 is fixed at row 1. Because the output track of a module spans two channels, the output of INBUF1 can be routed to channel 5 where it can be connected to the input of the inverter INV1.

The output of INV1 again spans two channels so that it can be routed to channel 3. However, because the input of OUTBUF1 can span only one channel (to channel 2), there is a gap. Therefore, an LVT is needed to close this gap. In reality, APR placed the INV1 at row 0 instead of row 4, and the LVT spans from channel 0 to channel 8.

In path 3, because an additional buffer is used, this buffer can close the gap so that no LVT is needed.

Path 2 does not have any fixed pin. In this case APR has used two adjacent pins for the input (pin 67) and output (pin 68) to minimize the delay.

Figure 3-50 shows the delays which are analyzed by the Timer.

Figure 3-50. Path Delays as Analyzed by the Timer

```
;  1st  longest path to all endpins
; Rank Total Start pin   First Net   End Net      End pin
;  0   33.1  INBUF1:PAD   IIN1        OUT1         OUTBUF1:PAD
;  1   32.9  INBUF3:PAD   IIN3        OUT3         OUTBUF3:PAD
;  2   28.1  INBUF2:PAD   IIN2        OUT2         OUTBUF2:PAD
; 3 pins
;  1st longest path to OUTBUF1:PAD (rising) (Rank: 0)
; Total Delay  Typ Load Macro     Start pin      Net name
; 33.1   8.5   Tpd   0  OUTBUF    OUTBUF1:D      OUT1
; 24.6   5.9   Tpd   1  INV       INV1:A         IOUT1
; 18.7  18.7   Tpd   1  INBUF     INBUF1:PAD     IIN1
;  1st longest path to OUTBUF3:PAD (falling) (Rank: 1)
; Total Delay Typ Load Macro      Start pin      Net name
; 32.9   9.8   Tpd   0  OUTBUF    OUTBUF3:D      OUT3
; 23.1   5.5   Tpd   1  BUF       BUF3:A         IOUT3
; 17.6   6.8   Tpd   1  INV       INV3:A         IBUF3
; 10.8  10.8   Tpd   1  INBUF     INBUF3:PAD     IIN3
;  1st longest path to OUTBUF2:PAD (falling) (Rank: 2)
; Total Delay Typ  Load  Macro    Start pin      Net name
; 28.1   9.9   Tpd   0  OUTBUF    OUTBUF2:D      OUT2
; 18.2   7.4   Tpd   1  INV       INV2:A         IOUT2
; 10.8  10.8   Tpd   1  INBUF     INBUF2:PAD     IIN2
```

Of the three signal paths, path 1, has the longest delay (Rank 0) due to the LVT. Path 3 is slightly faster although it contains one buffer more. Path 2 is the fastest thanks to the automatic pin assignment.

The additional delay due to the use of a LVT in path 1 can be obtained by breaking down the total path delay into its delay elements. The delay for the hard macro INBUF from INBUF1:PAD to net IIN1 is 18.7ns. In path 2 and

path 3, this delay is 10.8 ns. This translates to an additional delay of about 8 ns for the LVT.

Figure 3-51 shows the floorplan for the TPC1020 in an 84-pin package.

Figure 3-51. TPC1020 Floorplan

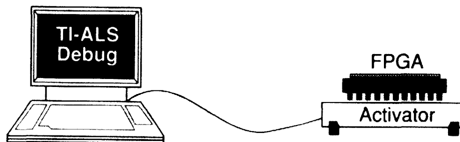
In most cases, TI FPGAs can be placed and routed totally automatically by the TI-ALS place and route software. However, if you want to use advanced features like manual placement control, timing optimization, etc., utilities exist which help you perform manual placement. You may also control timing through assignment of criticality to nets.

3.10 Using TI-ALS Debug†

3.10.1 Introduction

The TI-ALS has a Debug option which allows one hundred percent observability of every internal signal node while testing the FPGA in the Activator. This complements the Actionprobe which also allows one hundred percent observability of every signal node while the FPGA is in the target board (Figure 3-52).

Figure 3-52. FPGA Debugging



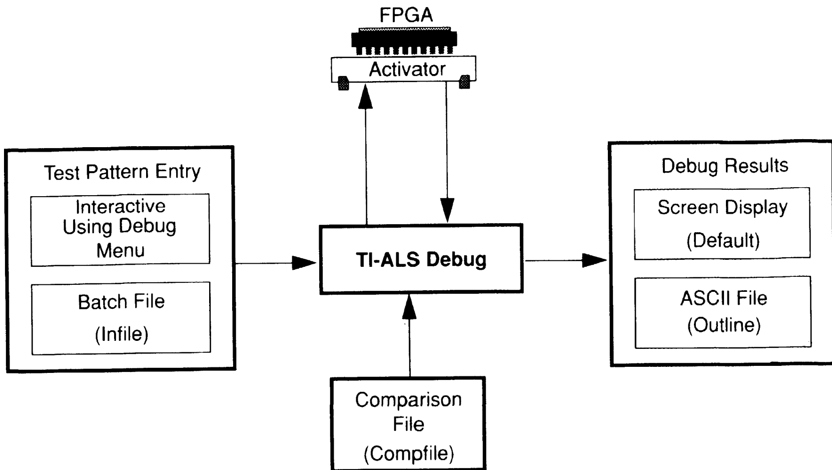
3.10.2 Functional Debug

The FPGA Debug option is a functional test to verify that the chip operates correctly once programmed. From within the TI-ALS environment, using the Debug menu option, you can force the state of the input pins and examine the response of observed output pins. The observed outputs are determined by the programmed device, rather than through a simulation model.

Figure 3-53 illustrates the Debug flow. The debugger is driven by commands which can be typed in from the keyboard, selected from a menu, or read from a batch file (infile). These three types of inputs for the Debugger allow the application of test vectors to the programmed FPGA.

† Contributed by Peer Uhlemann, FPGA Applications, Texas Instruments Deutschland GmbH.

Figure 3-53. Debug Flow



The Debug results are displayed on the screen or can be stored as an ASCII file and these results compared with the expected output values using a comparison file (`compfile`). This file can be generated from a modified simulation output file. The following are prerequisites for the Debug process.

- The tools work only on programmed devices.
- Program and probe security fuses must be unprogrammed.
- Pins SDI and DCLK are not available as user I/Os during Debug. The input to your circuit from these pins will be a logic 0 while you are in Debug mode.
- Pins PRA and PRB are not available as user I/Os during Debug.

3.10.3 TI-ALS Debug Menu

After logging into the TI-ALS system by typing **`als`** ***`design_name`*** and then **`debugger`**, the following submenu appears as shown in Figure 3-54.

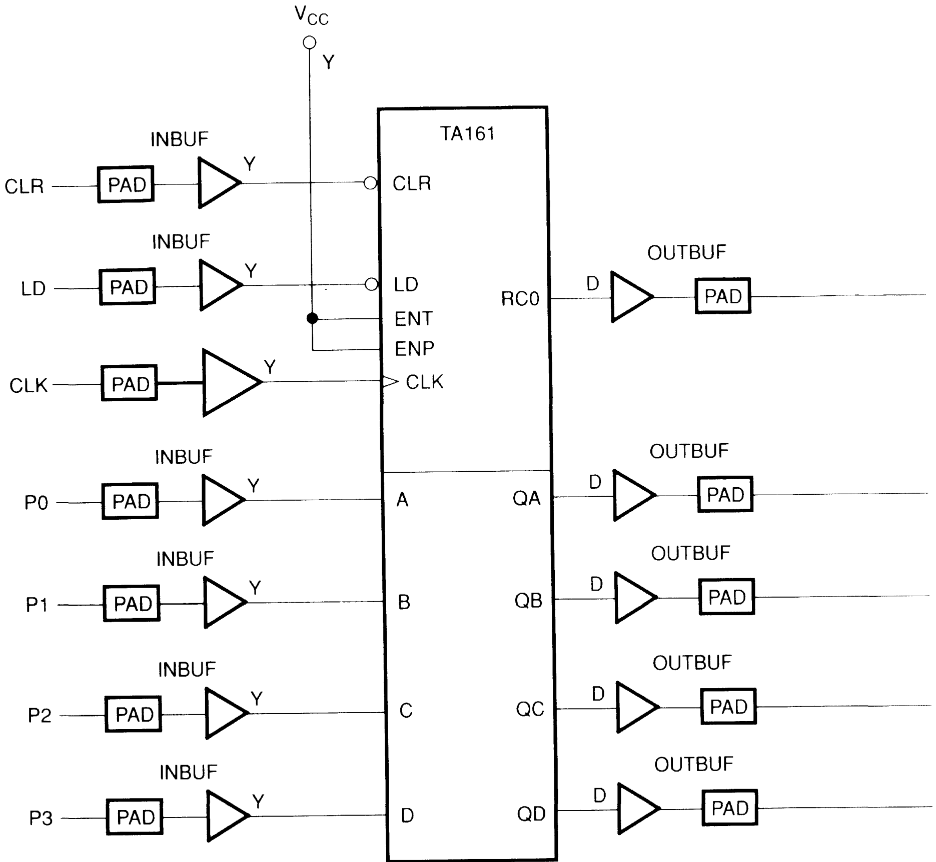
Figure 3-54. TI-ALS Debug Menu

Debugger	
Assign	Set a signal or a vector to a specific value
Low	Set a signal or vector to low
High	Set a signal or vector to high
Hi-Z	Set a signal or vector to 3-state
FAssign	File Assign: Load Input Stimulus
Step	Execute the commands previously specified
Print	Print the current value of the nodes to the screen
FPrint	Print the current values of the nodes to a file
Setup	Selects: Stimulus Vector Input File, Expected Responses Input File Output File, Load File
ICP	In-Circuit Probe: Invokes In-Circuit Probe
View	Scroll screen
Exit	Exit to ALS Main Menu
Design cnt4	Current design name cnt4

3.10.4 Counter TA161

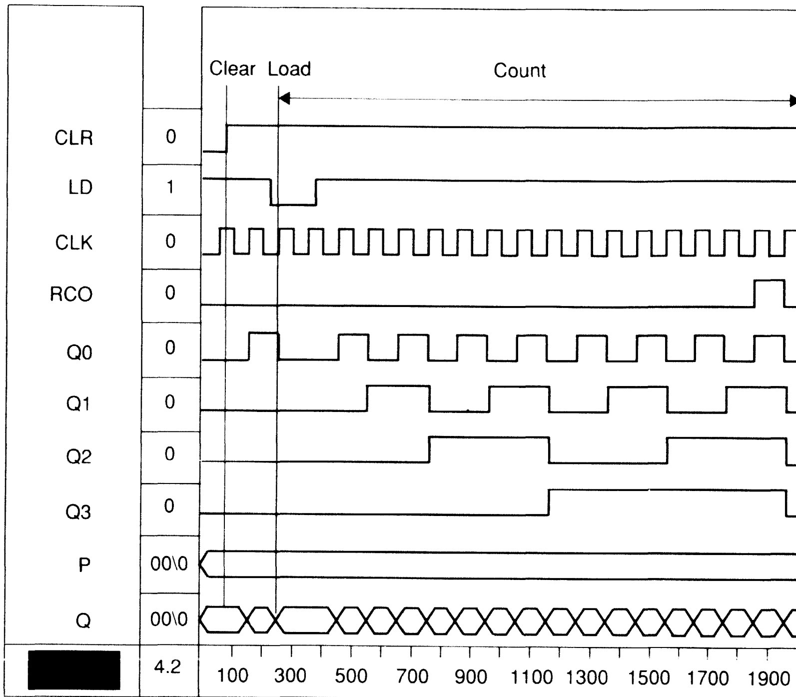
The TA161 is a synchronous 4-bit counter with preset and clear. This counter is available as a soft macro in the FPGA library. A soft macro is a predefined block consisting of multiple hard macros and/or other soft macros. Figure 3-55 shows the schematic of the 4-bit counter.

Figure 3-55. 4-Bit Counter TA161



The cascading inputs ENT and ENP are connected with V_{CC}. The load input LD is used to load a preset number (P0 to P3) to the counter. A sequence of clear, load, clock, and outputs is shown in Figure 3-56.

Figure 3-56. TA161 Waveforms



The counter does the following:

- Clear outputs to zero
- Load to a binary zero
- (P0=0, P1=0, P2=0, P3=0)
- Count up to the outputs and reset

3.10.5 Creating a Command File

A command file is the simplest way to apply a large number of command sequences to a device without having to retype the commands. Command files can be created using any ASCII text editor, such as Viewtext, but Debug commands in a command file must be enclosed in parentheses. Alternatively, commands can be entered from the Debugger menu. After creation of a command file, the file can be run while in the Debugger by selecting `Setup Loadfile` and entering the name of the command file. A

sample command input file is shown in Figure 3-57. The line numbers in the first column are included only for reference purposes.

Figure 3-57. Command File `cnt4.deb`

```
1 (emit "DEBUGGER DEMO FOR CNT4 DESIGN")

2 (vector P P3 P2 P1 P0)

3 (vector Q Q3 Q2 Q1 Q0)

4 (tabadd CLR LD CLK P Q RCO)

5 (infile "/designs/cnt4/cnt4.pat")

6 (outfile "/designs/cnt4/cnt4.res")

7 (compfile "/designs/cnt4/cnt4.cmp")

8 (define (clock) (l CLK) (step) (fprint) (print) (h CLK) (step)

(fprint) (print) (fcomp Q))

9 (define (clear) (l CLR) (clock) (h CLR) (clock))

10 (define (load) (l LD) (fassign P} (clock) (h LD) (clock))

11 (clear)

12 (load)

13 (repeat 15 (clock))
```

Explanation of example file in Figure 3-57:

Line 1: Prints the text to the screen. The text must be enclosed in double quotes.

Line 2: **vector** groups the load input bits to a vector P.

Line 3: Group the counter outputs to a vector Q.

Line 4: The **tabadd** command causes the nodes or vectors (CLR, LD, CLK, P, Q, RCO) to be displayed or printed when the **print screen** or **fprint file** commands are executed.

Line 5: **infile** opens an input file `cnt4.pat` containing input stimulus which can be applied to the device with the **fassign** command.

Line 6: **outfile** opens a file `cnt4.res` (Figure 3-58) for writing Debugger results with **fprint** command. The Debugger outputs resulting data in tabular form.

Line 7: **compfile** opens a file `cnt4.cmp` (Figure 3-59) used with the **fcomp** command. The file contains the expected output values which will be compared with the actual output values.

Line 8: The **define** command create a user macro clock which provides a pulse to the CLK input and prints all of the nodes to the outfile `cnt4.res`.

Line 9: Defines user-macro, clear, which clears the counter.

Line 10: Defines user-macro, load, which reads a load vector P from the input vector file `cnt4.pat` and loads the counter.

Line 11: Executes the user macro clear.

Line 12: Executes the user macro load.

Line 13: Repeats the macro clock 15 times, which causes the counter to count for 15 cycles.

Figure 3-58. Example Outfile `cnt4.res`

```
S          C L C P     Q    R
T          L D L            C
E          R   K            O
P
00001  : 0 1 0 1111 0000 0
00002  : 0 1 1 1111 0000 0
00003  : 1 1 0 1111 0000 0
00004  : 1 1 1 1111 0001 0
00005  : 1 0 0 0000 0001 0
00006  : 1 0 1 0000 0000 0
00007  : 1 1 0 0000 0000 0
00008  : 1 1 1 0000 0001 0
00009  : 1 1 0 0000 0001 0
00010  : 1 1 1 0000 0010 0
00011  : 1 1 0 0000 0010 0
00012  : 1 1 1 0000 0011 0
```

```
00013  : 1 1 0 0000 0011 0
00014  : 1 1 1 0000 0100 0
00015  : 1 1 0 0000 0100 0
00016  : 1 1 1 0000 0101 0
00017  : 1 1 0 0000 0101 0
00018  : 1 1 1 0000 0110 0
00019  : 1 1 0 0000 0110 0
00020  : 1 1 1 0000 0111 0
00021  : 1 1 0 0000 0111 0
00022  : 1 1 1 0000 1000 0
00023  : 1 1 0 0000 1000 0
00024  : 1 1 1 0000 1001 0
00025  : 1 1 0 0000 1001 0
00026  : 1 1 1 0000 1010 0
00027  : 1 1 0 0000 1010 0
00028  : 1 1 1 0000 1011 0
00029  : 1 1 0 0000 1011 0
00030  : 1 1 1 0000 1100 0
00031  : 1 1 0 0000 1100 0
00032  : 1 1 1 0000 1101 0
00033  : 1 1 0 0000 1101 0
00034  : 1 1 1 0000 1110 0
00035  : 1 1 0 0000 1110 0
00036  : 1 1 1 0000 1111 1
00037  : 1 1 0 0000 1111 1
00038  : 1 1 1 0000 0000 0 .
```

Figure 3-59. Example Comparison File `cnt4.cmp`

```
0b0000
0b0001
0b0000
0b0001
0b0010
0b0011
0b0100
0b0101
0b0110
0b0111
0b1000
0b1001
0b1010
0b1011
0b1100
0b1101
0b1110
0b1111
0b0000
```

3.11 Actionprobe for Functional Debug†

Section 3.11 describes the unique features of the TI-ALS Actionprobe. The Actionprobe allows one hundred percent observability of internal nets in a programmed TI FPGA device operating in the target system under its normal conditions. The following text in this section describes how the Actionpprobe is configured within the Debug environment of the TI-ALS development system and how the hardware is connected.

3.11.1 Introduction

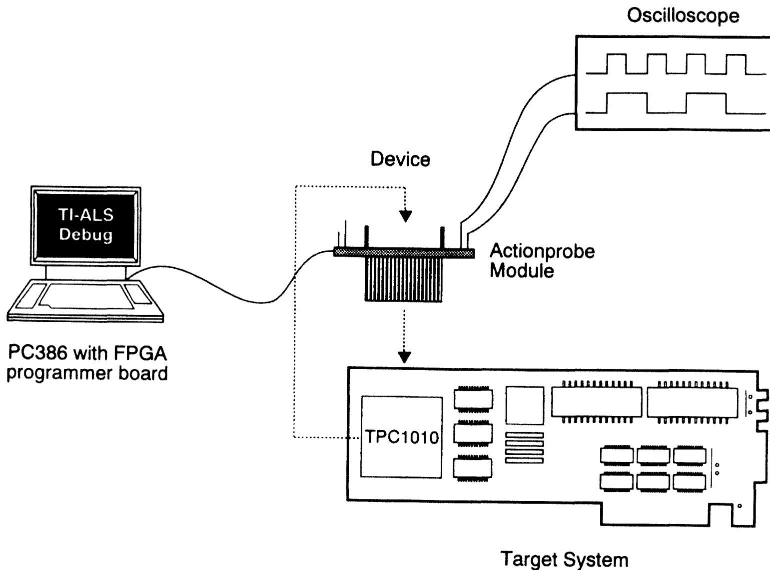
The Actionprobe is a unique feature of the TI FPGA architecture. With this system you can observe internal nets toggling up to 10 MHz for TPC10 devices and up to 50 MHz for TPC12 devices. This is achieved by programming the FPGA in the normal way and inserting the device into the Actionprobe module before the Actionprobe is plugged into a socket in the target system. Two external pins, PRA and PRB, are controlled from the TI-ALS Debug menu. This allows the selected internal nets to be observed on the pins.

3.11.2 Actionprobe Setup

Figure 3-60 shows the setup for using the Actionprobe for in-circuit diagnostics. The TI-ALS development system, Actionprobe module, target system, programmed TI FPGA, and an oscilloscope or logic analyzer are required to perform in-circuit probe functions.

† Contributed by Doug Mackay, Technical Marketing, Texas Instruments Ltd.

Figure 3-60. Actionprobe Setup



The programmed FPGA is inserted into the Actionprobe module and the module is inserted into the target system. The system will still function normally because the module will appear transparent. A serial link connects the module to the TI-ALS system which is used to interface to you; you will be prompted for the internal net names. Once the nets have been programmed to the package pins, an oscilloscope is used to observe the waveforms. These waveforms directly represent the net chosen, but they will be inverted. The Actionprobe module has test pins for all of the device pins, thus allowing easy connection for observation.

It is not recommended to perform precise timing measurements with the probes because small delays will be inherent in routing the signal to the package pins.

3.11.3 Configuring the Actionprobe

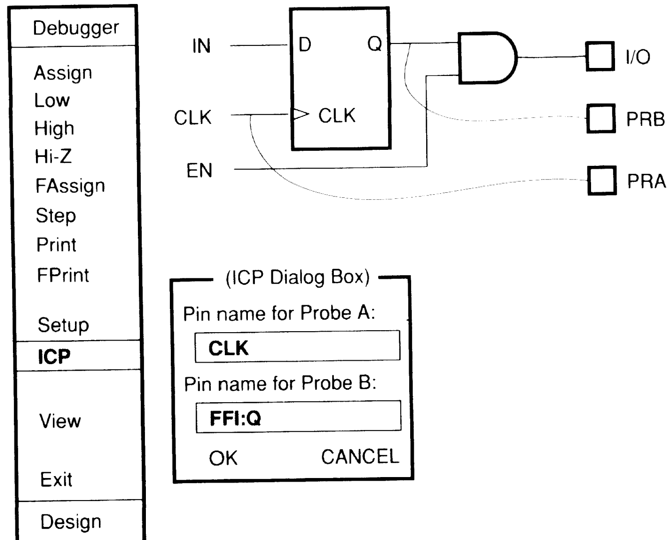
From within the Debug menu of the TI-ALS development system the in-circuit probe (ICP) utility is selected. This brings up a small dialog box. You are prompted for the logic module pin name or internal net name for probe A and probe B. Figure 3-61 shows the menu entry for a simple D-type flip-flop example.

Net CLK is selected for probe A and pin FF1:Q is selected for probe B. Selecting the OK from the dialog box will now program the FPGA device via the Actionprobe module situated in the target system. This takes about a second to program and can be reprogrammed an infinite number of times.

Note:

Here, *programming* means that the device is configured for debug. Do not confuse this with the actual programming of the device antifuse.

Figure 3-61. Configuring the Actionprobe



3.11.4 Actionprobe Resources

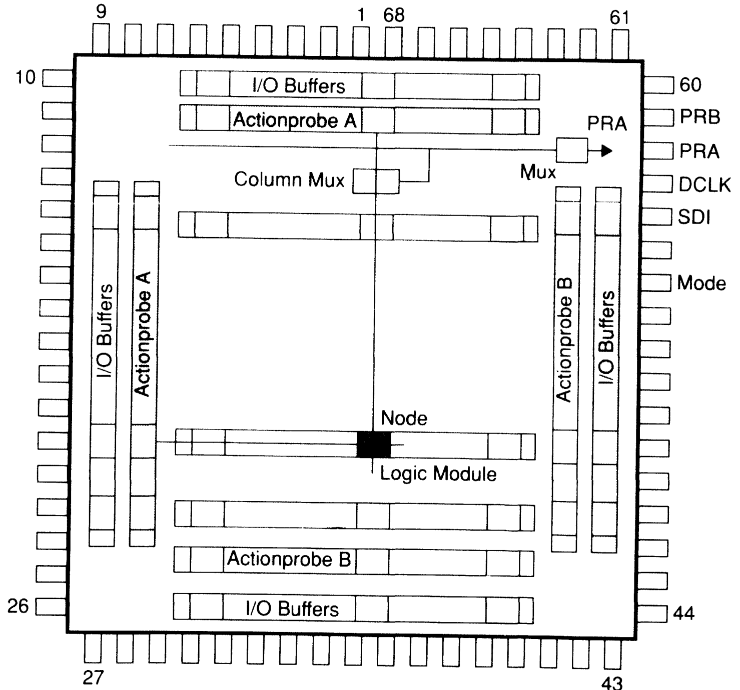
To obtain this unique feature of total observability in the target system five device package pins are required to carry out the necessary setup and signal observation. If the features of Actionprobe are not required then four of the five pins can be configured as inputs or outputs. This is shown in Table 3-1.

Table 3-1. Device Package Pins and Assignments

Pin	Function	Non-Actionprobe Assignment
PA	Probe A	Input
PRB	Probe B	Input
SDI	Serial data input	Output
DCLK	Data clock for SDI	Output
MODE	Selects user or debug mode	GND

A 10K pulldown resistor should be used on the PCB for SDI, DCLK and mode pin when using Actionprobe. Figure 3-62 shows a diagram of the location of the pins for a 68 PLCC device. This figure also shows the internal registers used to store the address of the selected net loaded via the SDI pin.

Figure 3-62. Resources Used for 68 PLCC Device



Each probe pin has a row and column register which stores the address for the internal node selected. A sequence of multiplexers is then used to channel the internal node to the package pin for observation by an oscilloscope or logic analyzer. As the address is stored in internal registers, the connection can be removed from the TI-ALS system where the programmed net address will still be retained in the FPGA device.

The unique features of the TI-FPGA allow total observability after the FPGA is programmed and working in the target system. This invaluable debug tool is used when the FPGA is inserted into the target system and, due to unforeseen conditions in the external environment of the FPGA, the system functions incorrectly. An example of this would be asynchronous switching signals that would be impossible to emulate on a simulator.

Debugging this type of problem is very time consuming and could take you days and weeks to find the problem and fix it. With the Actionprobe, you can find and correct the problem in a matter of hours, saving valuable development time and allowing the end product earlier entry into the market.

3.12 Net Criticality within Workview[†]

3.12.1 Introduction

Net criticality assignment is an important control of sensitive parts in a design because it gives the place and route tools additional information for optimization during the physical design process. Net criticality minimizes the wire lengths on nets between the macros during execution of the place and route algorithm.

Normally criticality assignment is done outside of the ALS environment by editing the `design_name.crt` file. This file is created by generating an ADL netlist, and is empty for you to fill in. In this section, you will learn how criticality assignments can be made with the Viewlogic schematic capture tool.

3.12.2 Levels of Criticality

TI-ALS supports four levels of net criticality:

- Fast criticality

  Should be reserved for the most speed critical nets in the design. The fast criticality assignment (CRT=F) causes the place and route algorithm to use the fastest routing tracks in the design to connect macros on shortest nets. A fast critical net will not be routed on a long vertical or horizontal track. Do not assign it to more than six percent of all the nets in the design for it to be effective.

- Medium criticality

  Specify (CRT=M) nets to a lower critical level and the router will attempt to keep to the median value for fan-out on the net. A critical net will not be routed on a long vertical or horizontal track. Because long routing tracks exhibit longer-than-average delays. The sum of critical and fast critical nets should never exceed twenty percent of the total nets in the design.

- Default criticality

  These are nets of minor concern. If a net is left unspecified, it is assumed to be default critical.

[†] Contributed by Peer Uhlemann, FPGA Applications, Texas Instruments Deutschland GmbH.

- Uncritical assignment

  Reserved for nonspeed critical nets, uncritical assignment (CRT=U) allows the placement and routing algorithm to use a long vertical or horizontal track. They are not speed-dependent and typically, uncriticality is used for reset or enable signals.

The Validator will give an error and warning message if the number of critical nets in the design exceeds the following percentages:

Table 3-2. Criticality Levels and Validator Errors and Warnings

Criticality Level	Warning	Error
Fast critical	if > 3%	if > 6%
Critical + fast critical	if > 15%	if > 21%
(6% of nets must be specified as uncritical)		
Critical + fast critical – uncritical	if > 10%	if > 15%

3.12.3 Define Critical Net

1) Select the net concerned with the mouse in Viewdraw.
2) Use Viewdraw menu `Add | Attr` and execute with the middle button.
3) Viewdraw prompts you for the attribute text string:
   - `CRT=F`—specify fast critical nets
   - `CRT=M`—specify critical nets
   - `CRT=U`—specify uncritical nets

Criticality may also be assigned to the internal nets of soft macros and user-defined macros. Too many critical nets will influence the efforts of the autorouter and the design can become unroutable.

Chapter 4

FPGA Logic Synthesis and Modeling

This chapter presents the FPGA logic synthesis and modeling tools. Schematic entry has been the only tool for electronics designers to design an electronic circuit or system; but, during the last few years, logic synthesis and modeling have become very attractive alternatives.

4.1 Logic Synthesis for FPGAs†

4.1.1 Introduction

This section describes the design methodology trend and the available logic synthesis tools for FPGAs on a PC or a workstation.

4.1.2 Design Methodology Trend

During the last 15 years the methodology of electronics design has undergone significant change. In the 1970's, PCBs with a handful of TTL ICs, each with a density of 10–100 gates, were state-of-the-art. The most popular piece of equipment in the design lab was the oscilloscope. CAD tools and simulation were nearly unknown to the typical designer.

In the early 1980's, a product called Programmable Array Logic (PAL) was introduced. This product has revolutionized the logic design methodology due to its user-programmability. It was the first time that the average designer encountered logic synthesis; that is, describing a circuit using Boolean equations or state machine entry and then running a compiler to generate a fuse file used to program the PAL device. There are good PAL/ Programmable Logic Device (PLD) synthesis tools which are supported either by the PLD vendors or third parties. Simulation is not necessary for PLDs because the devices have low complexity and the timing is normally not critical.

Also during the 1980s, application specific integrated circuits (ASICs) appeared and ASIC design methodology became an essential tool to design complex systems. Due to the need to verify the functionality of a design, especially the timing, simulation became one of the most important steps of the design flow. A computer aided design (CAD) ASIC design environment typically has a good simulator tightly coupled with a powerful schematic entry tool. This hardware-oriented gate-level design approach provides an experienced designer good visibility into his design. He can modify a node or a net and simulate to see what happens with the signal.

Today an ASIC exists with 150K gates or more on a single chip. At this level of complexity, it is almost impossible to do a design without sophisticated tools because it is not only time consuming to design the circuit, but also difficult to document or modify the design. These are some of the reasons why high-level hardware description languages (HDL) and very high scale integrated circuit (VHSIC) hardware description languages (VHDL) are becoming the tools of choice for high-gate-count designs.

† Contributed by Dung Tu, Technical Marketing, Texas Instruments Deutschland GmbH.

On the highest level, HDLs allow description of a circuit as behavioral models.A design compiler can synthesize, optimize and map the design to a specific technology library. Design with HDL is comparable to programming with a high-level language while design with schematic entry is similar to programming with a machine code. The productivity increase can be a factor of five or more.

Field programmable gate arrays are high-density programmable logic devices which are intended to build a bridge between PLDs and classic ASICs. Small FPGAs are used for production while the more complex FPGAs are useful for protoptyping a design which can be built later in a gate array or standard cell.

Due to its complexity, the smallest FPGA can replace 10 PLDs. FPGA designs should be simulated so that the timing can be verified. That is why FPGA design kits are normally delivered with a simulator and a schematic entry tool. On the other hand, there are many PLD designers who have experience with PLD logic synthesis and wish to use a similar approach for FPGAs. For these designers, gate-level design is a step backward. Another important reason for logic synthesis is the need to migrate an FPGA to an ASIC.

4.1.3 FPGA Logic Synthesis Tools

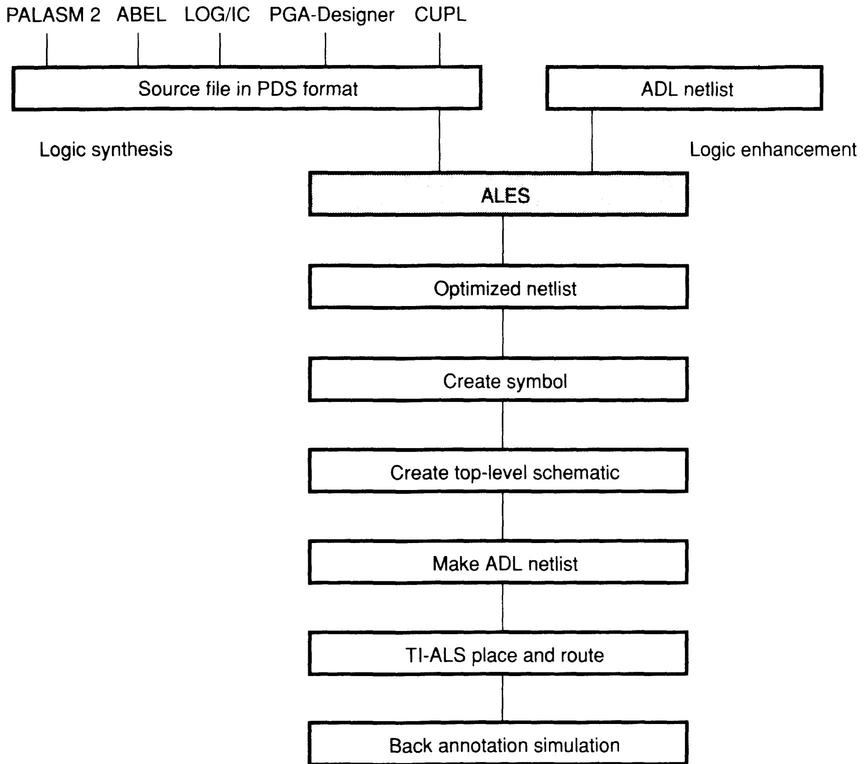
There are two criteria for an FPGA logic synthesis tool. First, It must allow the PLD designer to use his familiar, existing PLD tools on FPGAs, and second, it must be powerful enough to handle the migration to ASICs.

In the market for high end logic synthesis tools, Synopsys™, Cadence™, and Mentor are the main players. FPGA logic synthesis is now supported for all. The most popular PLD design tools at the low end are ABEL™ (Data I/O), LOG/IC™ (Isdata), PGADesigner™ (Minc), proLogic™ (TI), PALASM™ 2 (AMD), and CUPL™ (Personal CAD Systems). Except for the PGADesigner and the new ABEL FPGA, these tools do not directly support FPGAs; however, for historical reasons, the PAL device design specification (PDS) format of PALASM 2 has become a defacto standard for PLDs and the other PLD tools have an option to generate the `.pds` format. Many synthesis tools can take a `.pds` file as input, synthesize the logic, and generate an optimized FPGA netlist.

4.1.4 FPGA Logic Synthesis

The logic synthesis design flow is shown in Figure 4-1.

Figure 4-1. FPGA Logic Synthesis Flow



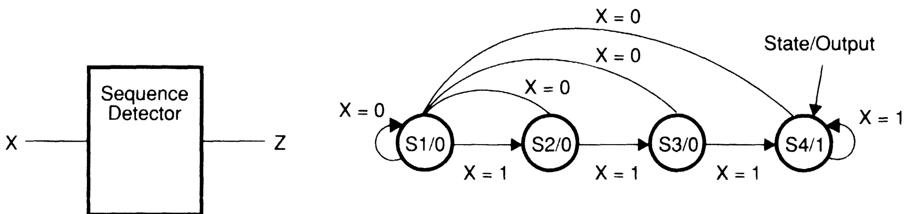
The PDS source file for a functional block which is to be synthesized can be entered as an ASCII file, using the PALASM 2 syntax, or created by one of the PLD tools. Synthesis generates a netlist for the block. In addition, instead of a `.pds` file, many tools take an `.adl` netlist as input for the synthesis.

The next steps are to create a symbol for the block and a schematic for the top level. The top-level schematic can consist of several synthesized blocks, mixed with normal FPGA hard macros or soft macros. The top-level schematic is then used to generate an advanced design language (ADL) netlist for place and route as well as a wirelist so that back annotation simulation can be done after place and route.

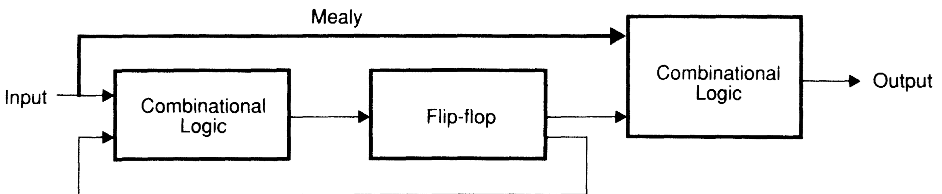
A classic text book example for a simple state machine is a sequence detector. This circuit monitors an input signal X and outputs a signal Z=1 when the input X is X=1 for three successive clock cycles.

Figure 4-2 illustrates the state diagram for the sequence detector, designed as a Mealy state machine. The initial state is S1 and the output is Z=0 (S1/0). If X=1, the machine moves to the next state, otherwise it stays in the same state. When in state S4 the output is Z=1, because X has been X=1 for three successive clock cycles.

Figure 4-2. State Diagram Sequence Detector



- The sequence detector monitors an input signal X and outputs a signal Z = 1 when X = 1 for three successive clock cycles.



- The state machine is implemented as a Mealy machine; that is, the output depends not only on the state but also on the inputs.

Figure 4-3 shows the source file for the sequence detector using LOG/IC as the design software to describe the state machine. LOG/IC has a powerful state table syntax which allows a very efficient description of a state machine.

The option PROGFORMAT = P-EQUATIONS generates the `.pds` file for the state machine. Note that LOG/IC has generated two signals QQ1 and QQ2, which automatically store the state bits. These two signals must be drawn on the symbol, but they are left open. The internal connection is included in the block netlist.

Figure 4-3. Source Files

a) Source File in LOG/IC format

```
*IDENTIFICATION
Mealy Sequence Detector
Dung Tu
Texas Instruments, FPGA Applications Europe
*X-NAMES
X;        !Input signal for detector
*Y-NAMES
Z;        !Output of detector
*FLOW-TABLE
;State #, X Input value, Y Output value, Following state#
S1, X1, Y0, F2; !The sequence detector counts
S1, X0, Y0, F1; !The input bit stream X. It

S2, X1, Y0, F3; !outputs a signal Z = 1 when
S2, X0, Y0, F1; !X = 1 for 3 successive clocks

S3, X1, Y0, F4;
S3, X0, Y0, F1;

S4, X1, Y1, F4;
S4, X0, Y0, F1;
*STATE ASSIGNMENT
BINARY;
*RUN-CONTROL
PROGFORMAT = P-EQUATIONS;
*END
```

b) Source File in PDS Format

```
TITLE  Mealy Sequence Detector
PATTERN
REVISION
AUTHOR  Dung Tu
COMPANY  Texas Instruments, FPGA Applications Europe
DATE  92/12/30  7:27:05

CHIP MEALY USER

CLK Z QQ2 QQ1 X

EQUATIONS

    Z = X     * QQ2   * QQ1

    QQ2  := X   * QQ1
          + X   * QQ2

    QQ1  := X   * QQ2
          + X   */QQ1
```

Figure 4-4 shows the top-level schematic which includes the symbol Mealy for the sequence detector and the input/output buffers.

Figure 4-4. Top-Level Schematic Including I/O Buffers

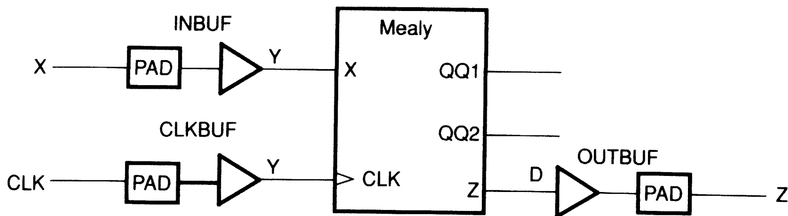
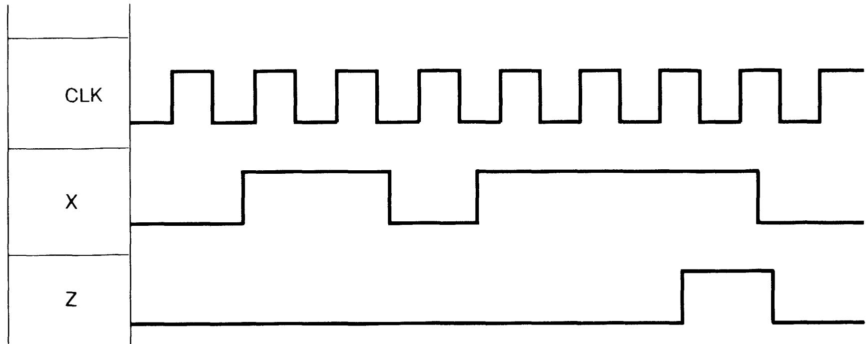


Figure 4-5 shows the result of the back annotation simulation.

Figure 4-5. Back Annotation Simulation



4.1.5 FPGA Logic Synthesis with Synopsys Using VHDL/Verilog™

Figure 4-6 illustrates the FPGA logic synthesis flow with Synopsys using VHDL or Verilog. The source file can be either in VHDL or in Verilog format. To map the design to FPGA, the Synopsys design compiler requires an FPGA library. The optimized design can be written out as an EDIF schematic or netlist. At this point there are two options.

The first option is to use an electronic design interchange format (EDIF) converter to convert the EDIF schematic either to a Viewlogic, Mentor, or Valid schematic. The rest of the design flow can be done with an appropriate FPGA development kit running on the same platform.

In the case of Viewlogic (the schematic conversion can be done with EDIF2VL2), the schematic can be transfered to a PC and the rest of the design flow can be PC-based.

The second option is to work with the EDIF netlist and use an EDIF reader to translate the EDIF netlist to Viewlogic, Mentor, or Valid netlist format. The **makeadl** program of the TI-ALS translates these netlist formats into ADL format.

Figure 4-6. FPGA Logic Synthesis Flow with Synopsys

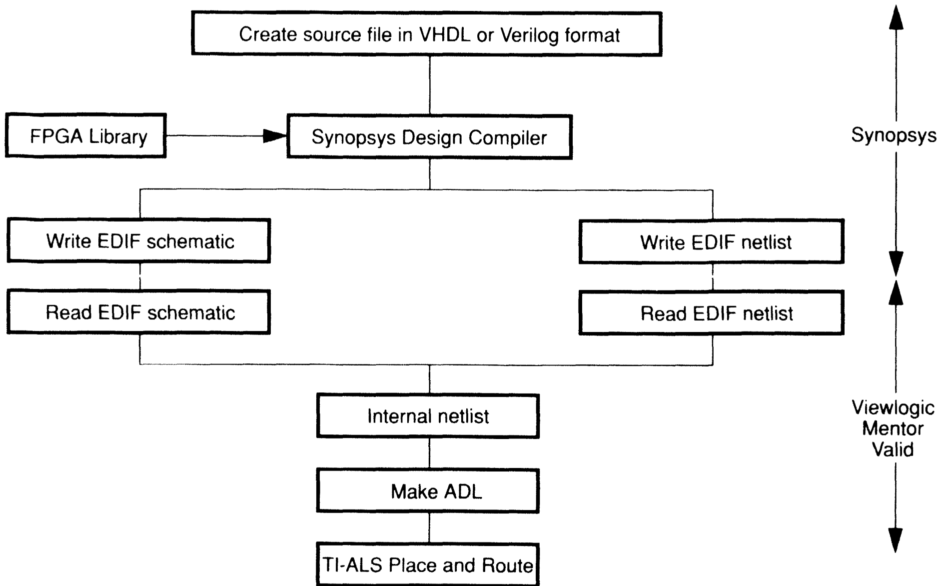


Figure 4-7 shows the VHDL source file and the script file for the same sequence detector as used to demonstrate the ALES 1 flow.

Figure 4-7. VHDL Source File, Synopsys Script File

a) VHDL Source File

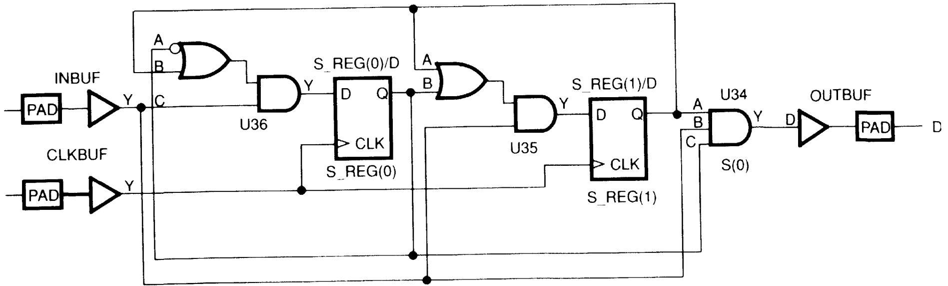
```
entity MEASEQ is
    port(X,CLK :in BIT;
         Z :out BIT);
end;
architecture BEHAVIOR or MEASEQ is
    type STATE_TYPE is (S1, S2, S3, S4);
    signal S, SNEXT: STATE_TYPE;
begin
    COMBIN:process
    begin
         case S is
             when S1 =>
                 if X = '1' then Z <= '0';  SNEXT <= S2;
                 else Z <= '0';             SNEXT <= S1;
                 end it;
             when S2 =>
                 if X = '1' then Z <= '0';  SNEXT <= S3;
                 else Z <= '0';             SNEXT <= S1;
                 end it;
             when S3 =>
                 if X = '1' then Z <= '0';  SNEXT <= S4;
                 else Z <= '0';             SNEXT <= S1;
                 end it;
             when S4 =>
                 if X = '1' then Z <= '1';  SNEXT <= S4;
                 else Z <= '0';             SNEXT <= S1;
                 end it;
             end case;
         end process;
         SYNCH: process
         begin
             wait until CLK event and CLK = '1';
             S <= SNEXT;
         end process;
     end BEHAVIOR:
```

b) Script File

```
/*** Design check ***/
designer       = "Dung Tu";
company        = "Texas Instruments";
search_path  = {./win1/ti/fpga
link_library = tpc10.db
target_library   = tpc10.db
symbol library   = tpc10.sdb
read -f vhdl measeq.vhdl
current_design = MEASEQ
write -h -f db -hierarchy -o measeq.db
check_design > chkdsn.rpt
report_design > measeq.rpt

/*** Constraint: Optimize for area***/
max_area 0.0
compile
write -h -f db -o measeq_opt.db
report -area -cell -timing > measeq.rpt
free all

/*** Write EDIF schematic***/
read -f db measeq_opt.db
current_design = MEASEQ
create_schematic -hierarchy -size B
write -h -f edif -o measeq.edif
exit
```

Figure 4-8 illustrates the schematic of the sequence detector which is synthesized by Synopsys. This example shows the usefulness of logic synthesis. Without this tool, the designer must use Karnaugh map technique to synthesize and optimize the design manually, then derive the equations for the flip-flops and translate these to a schematic.

Figure 4-8. Sequence Detector Schematic



4.1.6 Summary

Logic synthesis allows you to describe a design independent of the technology so that you can concentrate on real design issues like architecture, functional blocks, etc. Depending on which technology is desirable, you can map the design later on to an appropriate technology library, either FPGA, gate array, or standard cell. Due to the availability of good design tools and standardization, VHDL is becoming the hardware description language of choice, especially for designs of high complexity.

4.2 ALES 1—Logic Enhancer/Synthesizer Tool for PC-386†

This section discusses the use of the logic enhancer/synthesizer unit (ALES 1) on the automatic conversion of a PLD design into an TI FPGA using a PALASM 2 source file. The basic features of the tool are discussed and the design flow is described using a synchronous counter from a PALASM 2 file.

Note:

ALES 1 software was discontinued in January, 1993. New synthesis software, TPC-ALS-PLDSYN, and a technical primer are currently in development with a target release date of March, 1993. The information is this section is background data.

4.2.1 Introduction

Since the development of the PLD in the late 1970s, we have seen how one can integrate many general purpose logic (GPL) parts into one PLD and are now beginning to see the next stage in this field programmable logic (FPL) evolution; that is, the integration of multiple PLD products into a single FPGA. This brings the designer closer to having complete system level integration in a single FPL product. Because the TI FPGA architecture is different from conventional PLD architecture, the transition from PLD to TI FPGA is not straightforward and special design automation tools have been developed to make this transition fast and simple.

The logic enhancer/synthesizer unit (ALES 1) can automatically translate a PLD design into an FPGA netlist and optimize this design for speed and area. This allows PLD designers to continue to use their familiar design languages, state machines, and Boolean equations and also to integrate a PLD design into a system-level design captured from circuit schematics. It has another valuable benefit which allows an `.adl` netlist to be optimized for area or speed. This means that a design from schematic capture can be optimized to best utilize the TI FPGA architecture.

This section describes the basic features of the tool and its design flow followed by an application example using a 5-bit synchronous counter, which counts up to 24, designed from PALASM 2 equations.

† Contributed by Doug Mackay, Technical Marketing, Texas Instruments Ltd.

4.2.2 Function of ALES 1

ALES 1 has two main functions.

- First, it can synthesize Boolean equations and/or state machine functions into an optimized `.adl` netlist ready for place and route of the design using TI-ALS.
- Second, it can optimize an `.adl` netlist into a more efficient implementation for the FPGA architecture.

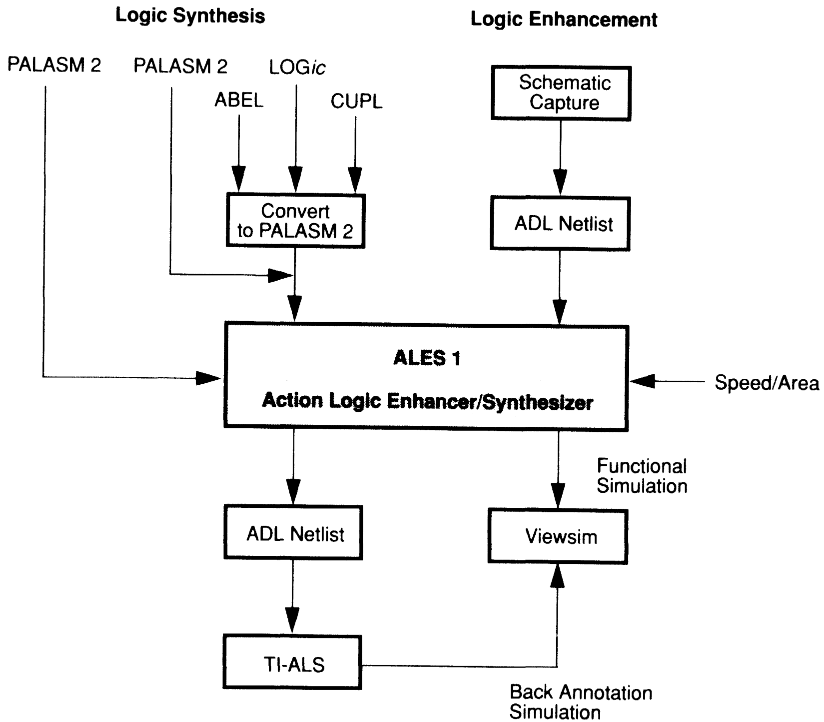
In either case, the designer has the option of minimizing delays or maximizing logic utilization using unique algorithms created for the TI FPGA architecture.

ALES 1 is targeted at PLD designers who need a tool to interface with the popular PLD design tools in the market. ALES 1 will accept source files, either individually or in combination, from PALASM 2, ABEL, CUPL, LOG/IC, PGADesigner, and the `.adl` netlist, allowing the designer to choose the best method for each portion of the design and simulate under the same environment within Viewlogic.

4.2.3 ALES 1 Design Flow

Shown in Figure 4-9 is both the logic synthesis and logic enhancement design flows. The logic synthesis flow accepts a textual description of the design in PALASM 2 form, in either state machine or Boolean equations. Once this has been generated or converted from ABEL, CUPL, LOG/IC or PGA Designer, ALES 1 is invoked with a switch which is set to optimize the design for area or speed.

Figure 4-9. ALES 1 Design Flow for PC



ALES 1 outputs an synthesized `.adl` netlist together with a Viewlogic `.wir` file, for the PC-386 version. The `.wir` file is used for simulation using the Viewlogic simulator, Viewsim. The `.adl` netlist produced can then be imported into part of a design or solely used for place and route from within the TI-ALS environment. This design flow will allow back annotating the extracted delays from the TI-ALS environment.

The logic optimization of an `.adl` netlist generated from schematic capture has a very similar design flow. A **makeadl** program, included with the TI-ALS system for each CAD package, converts the schematic netlist into an `.adl` netlist. This is then used as the input to ALES 1 with the switch set to optimize for area or speed. As for the PALASM 2 input, the output from ALES 1 is an optimized `.adl` netlist and a `wirlist` file for Viewlogic.

In addition to the area/speed option for ALES 1, there are several other switches that can be set, described more fully in Section 4.2.4.

4.2.4 ALES 1 Command Reference

Invoke ALES 1 with the following command:

`alesl [options]` ***design_name***

In addition to optimizing for area or speed, the following options are allowed when using ALES 1 on the PC-386 platform.

- -adl

  Reads a TI `.adl` file (logic enhancement)

- -pds

  Reads a PALASM 2 format file (logic synthesis)

- -family:<f>

  Specifies TPC10/TPC12 series family

  f = 1000 or 1200

- -del:<n>

  Attempts to make all paths < n ns delay

- -pr

  Preserves signals from original netlist

- -cpu:<n>

  Limits the CPU time to n mins,(default=max=60)

- -ncl

  Disallows combinational loops

- -cm

  Creates IO buffers automatically

- nobuff:<n>

  Inhibits buffering on internal net <n> if fan-out > 10 is detected.

More detailed information can be found in the *ALES 1 2.2 User's Guide.*

4.2.5 Application Example (5-Bit Synchronous Counter)

This section discusses the operation of ALES 1 showing the logic synthesis of a 5-bit synchronous counter PLD design. This counter was designed using PALASM 2 (PDS) descriptive language for a standard TI 16R6 PAL device.

4.2.6 Synthesis of a PALASM 2 Design

The PALASM 2 `.pds` file is an ASCII file which defines the functional operation of a programmable device. The syntax used is not case-sensitive and will disregard blank spaces and blank lines. Many of the keywords in PDS are not relevant to ALES 1, such as COMPANY, and are ignored. A full description of the syntax required for a `.pds` file for ALES 1 can be found in the *ALES 1 User's Guide*, Appendix A. The PALASM 2 source file used is shown in Figure 4-10 below and has been generated to implement a synchronous counter onto a 16R6 PAL device.

Figure 4-10. PALASM 2 Source File

```
;=============================================================
;    CNT24.PDS
;=============================================================
;
;    This file is a PALASM 2 Equation Syntax File of a 5-bit synchronous
                    counter which resets after counting to 24.
;
;=============================================================
;
;    Device declaration shows that a PAL of type 16R6 has been
;    chosen for this design.  A '/' is used for
;    asserted low signals.  The pin order is as per datasheet
;    for the PAL.  Pin 11 should be the enable signal which
;    is not used and shown connected to GND.
CHIP cnt24 PAL16R6
clk /clr nc nc nc nc nc nc nc gnd gnd nc qa qb qc qd qe nc nc vcc
;
;  Boolean registered equations are listed below.
EQUATIONS
qa := /qa;
qb := qa*/qb + /qa*qb;
qc := /qb*qc + /qa*qc + qa*qb*/qc;
qd := (/qc*qd + /qb*qd + /qa*qd + qa*qb*qc*/qd)*/qe;
qe := qa*qb*qc*qd*/qe + /qd*qe*(/qc+/qb+/qa);
;
;  The output extension .SETF is used to provide a Asynchronous
;  registered set
qa.setf = clr;
qb.setf = clr;
qc.setf = clr;
qd.setf = clr;
qe.setf = clr;
; =================== END OF PALASM2 FILE ===================
```

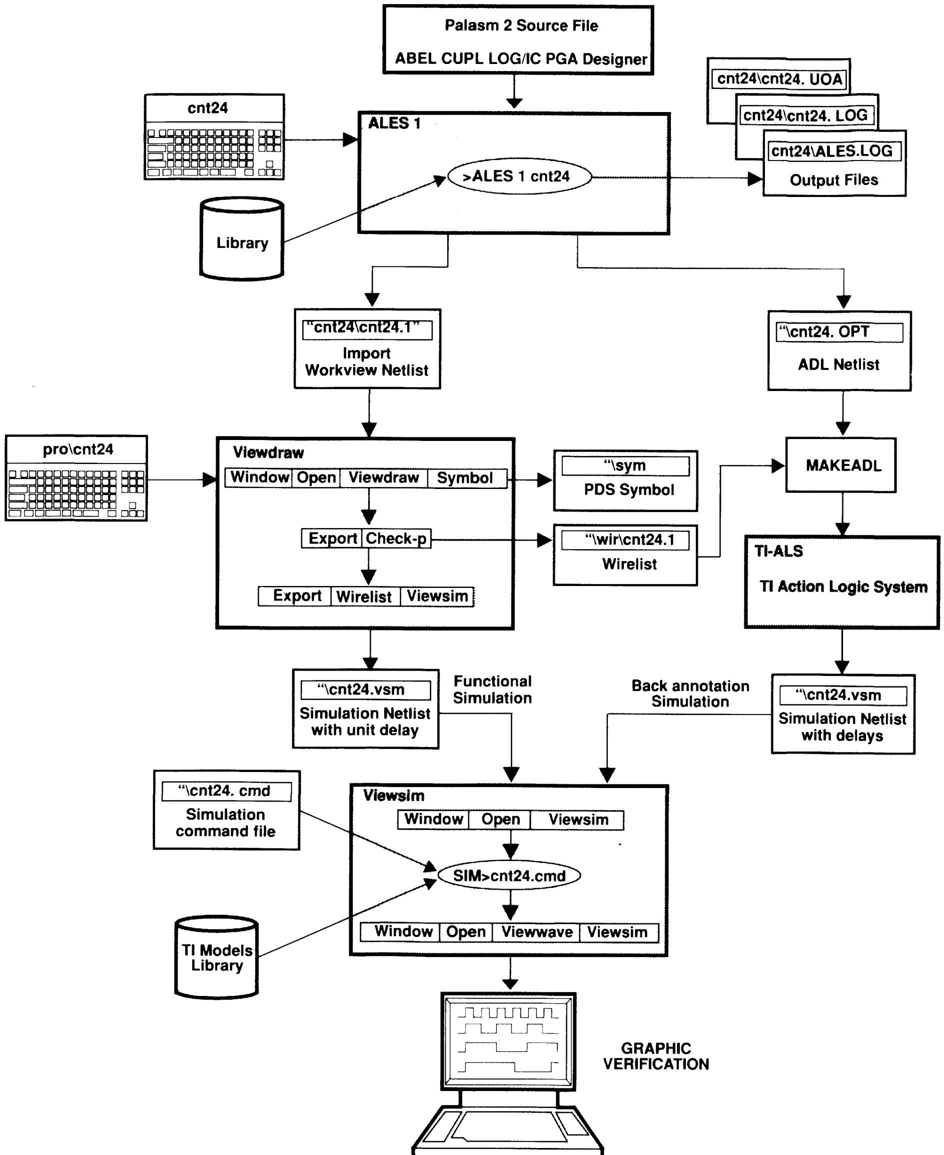
Figure 4-11 shows a graphic representation of the design flow used for this example. This design flow diagram shows the main tools used together with the files generated by each stage.

Having generated the necessary information for the source file, ALES 1 can be invoked with the required switches set. There is no need to modify the PDS file for using ALES 1. The reference to the 16R6 PAL device and its pin definition is handled by the software. For the 5-bit counter, CNT24, the command to execute ALES 1 is:

```
c:\designs> ales1 -lpds -del:20 -family:1000 cnt24
```

This starts ALES 1 for a PDS file `cnt24.pds` with the delay switch set to 20 ns. This attempts to make all delay paths in the design < 20 ns. The source file should reside under a directory in `\designs` with the `.pds` extension.

Figure 4-11. Logic Synthesis Design Flow



ALES 1 will output 4 files in the design directory and a Viewlogic wirelist in the `wir` directory. These files are:

`design.uoa` (an ADL netlist which is unoptimized)
`design.opt` (the resulting optimized ADL netlist)
`ales1.log` (shows all status, warning, and error messages)
`design.log` (gives a report of the ALES 1 program)

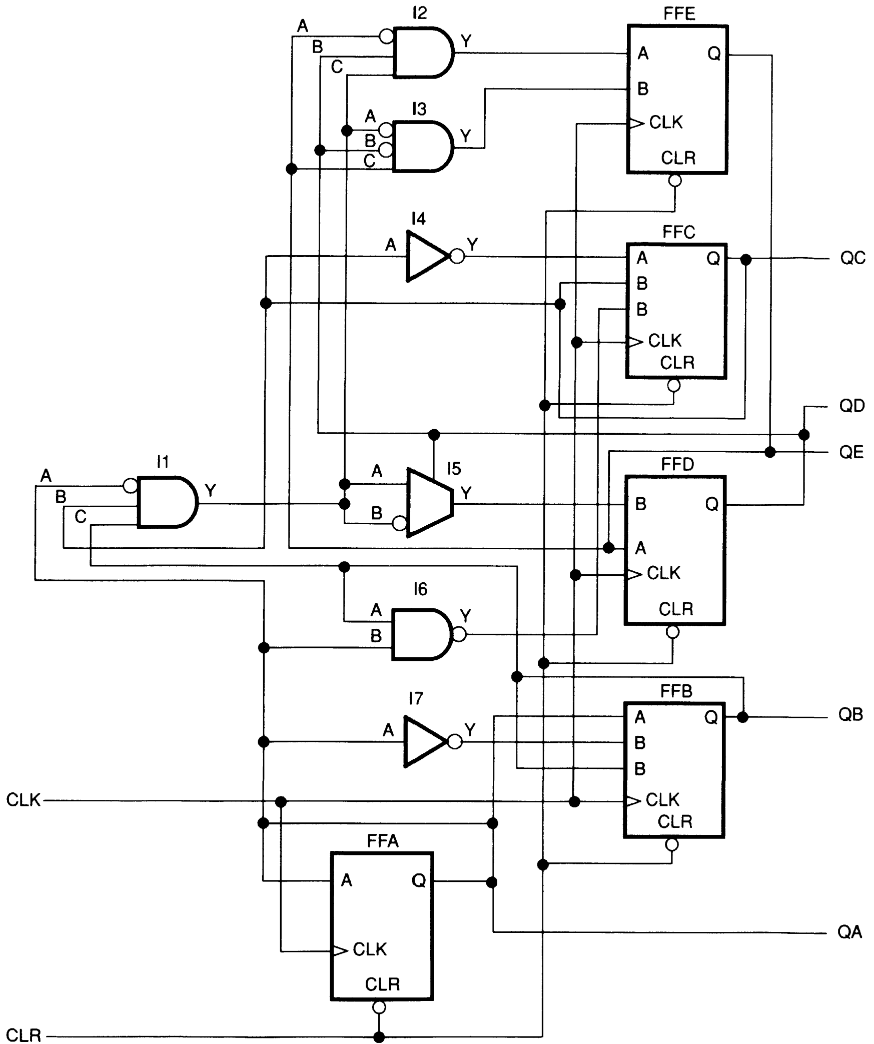
The log report shows all the necessary steps that occur in the conversion and optimization phases. When the original design is translated from equations you can see that there are 73 logic modules with a maximum input to output delay of 76 ns. This is then optimized with a result of 17 modules and 30 ns delay on the first pass. A total of six passes is executed in the attempt to meet the delay objective of 20 ns. If the objective is not reached, then the best result is saved (30 ns for this example).

The Viewlogic `.wir` files are then generated and saved under `d:\designs\wir` directory. The `.wir` file should be imported into the Viewlogic project area to verify the design by simulation. Before simulating, a symbol must be created and the unattached attribute, ALSPDS, must be added. This attribute informs the **`makeadl`** netlist converter not to convert the `wirlist` for this part of the design, but to use the `design.opt` netlist also generated from ALES 1. Once generated, this symbol can then be used in the top level of the design where input and output pads are added, or it can be integrated into other parts of the design. Simulating the resultant `.wir` file is done in the same way as if the design was created from the schematic.

Once the design is verified by prelayout simulation **`makeadl`** generates the ADL netlist required for place and route by TI-ALS. After delay extraction by TI-ALS, the last stage of verification before a FPGA device can be programmed is back annotation of the delays into Viewlogic and performing a postlayout simulation. An example of the PC-386 Viewlogic design flow is given in Section 3.1.

Figure 4-12 shows a graphic representation of the resulting `.wir` file from ALES 1 of the PALASM 2 design.

Figure 4-12. Counter Example



4.2.7 Optimizing an ADL Netlist

In addition to synthesizing a PAL into part or a full FPGA, ALES 1 also optimizes an `.adl` netlist for speed and area. This option is performed by first generating an `.adl` netlist using **makeadl**. ALES 1 is invoked with the source file and the -ADL switch and then uses an ADL netlist as a source file and outputs 4 files in the design directory and a Viewlogic wirelist.

4.2.8 Analyzing Results Obtained from ALES 1

From the simple 5-bit counter example used to describe the function of ALES 1, we can see substantial reductions in propagation delay and area internal to the TI FPGA.

The results of this example show increased logic minimization and device performance with a reduction of 13 logic modules. To obtain the same result from manual optimization would take a great deal longer.

4.2.9 Summary

This section on synthesizing PLDs into TI FPGAs has shown how easily ALES 1 can be used to take existing PLD designs and integrate them into a TI FPGA. This can be partial or total integration allowing the designer to design and simulate PLDs within a schematic representing the complete system.

The example shows how easy it is to use ALES 1 for logic synthesis for area or speed. Results show a saving of 43% in area and 30% increase in speed.

An interesting statistic obtained from the results is that seventeen similar 16R6 PAL designs can be integrated into a single TPC1010 device, the smallest TI FPGA in the family.

4.3 PLD, CPLD, and FPGA Design Flow Using PLDesigner-XL†

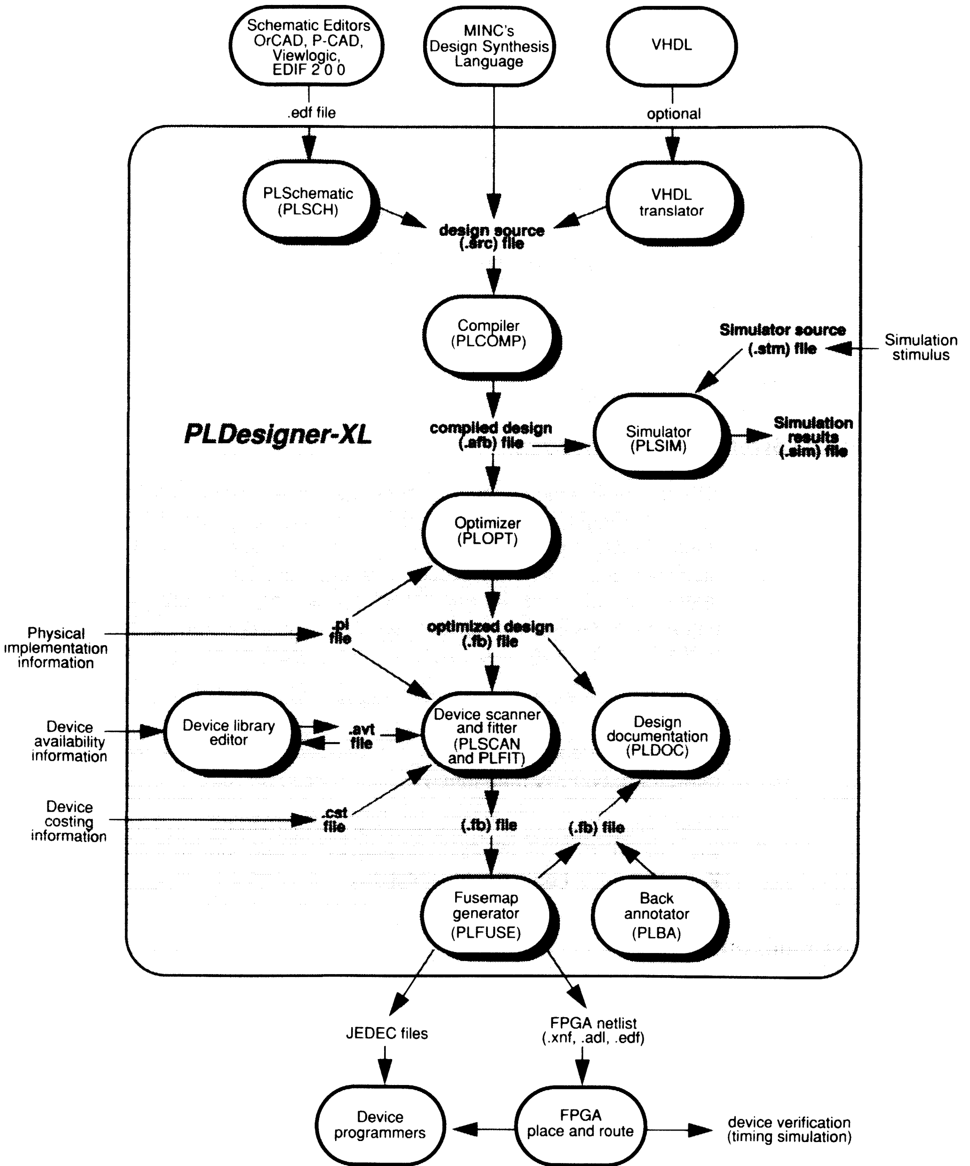
This section demonstrates how a design can be partitioned into multiple TI programmable logic devices using the PLDesigner-XL™. Figure 4-13 is a block diagram of the PLDesigner-XL system which shows the associated software tools that work with PLDesigner-XL. PLDesigner-XL is available on the PC, SUN, and HP platforms direct from MINC and on various other platforms from MINC's EDA OEM partners (Cadence, Mentor, Racal-Redac™, and Teradyne™/Sophia™).

4.3.1 Design Flow

Programmable logic design synthesis is the process of describing a design by schematic or language entry and synthesizing that information into an optimized form used to program one or more programmable devices. PLDesigner-XL is a full-featured programmable design synthesis tool that lets you concentrate on your design, not the operational details of the programmable devices. PLDesigner-XL synthesizes the path from design description to the actual programmable devices.

† Contributed by Mehrdad Banki, Applications Support Manager, Minc Incorporated.

Figure 4-13 . PLDesigner-XL Architecture



4.3.2 Design Entry

4.3.2.1 Flexible Design Methodology

PLDesigner-XL provides a device-independent approach that lets you enter your logic design without specifying devices for implementation. If desired, you may choose specific devices during design entry.

Design entry may be accomplished by one or a combination of methods. This feature allows you to describe each function using the entry method best suited for that particular function. The design entry methods include:

- Design synthesis language (DSL)
- Schematic entry
- EDIF 2 0 0 netlist entry
- VHDL (optional)

4.3.2.2 Design Synthesis Language (DSL)

DSL is a high-level behavioral language developed by MINC exclusively for use with programmable logic. DSL provides constructs for state machine descriptions, truth tables, and Boolean equations. DSL also allows hierarchical design with procedures and functions. Program control-flow statements such as IF and CASE, combined with multiple nesting and hierarchical design capabilities, let you describe complex designs quickly and easily. You may also create macros to perform text-substitution.

All designs entered using MINC's design entry methods are fully compatible with PLDesigner-XL on UNIX-based or DOS platforms. Designs created in the DOS or UNIX environments are also compatible with MINC's EDA OEM partners previously mentioned.

MINC also supports a synthesis library (PLDPRIMS) for Synopsys through EDIF 2 0 0, which gives Synopsys users the means of targeting Programmable Logic Device solutions.

4.3.3 Schematic Entry

The PLDesigner-XL supports direct schematic entry from OrCAD, P-CAD™, Viewlogic, and EDIF 2 0 0. Schematic support is also available from MINC's EDA OEM partners previously mentioned.

4.3.4 Device Selection

4.3.4.1 PLDs/Complex PLDs (CPLDs)

PLDesigner-XL automates the selection of the best TI PLD/CPLD device architectures for your design. Based on the device characteristics and your design constraints, the device selection system searches the master library for devices that match your constraints. Your design is then mapped into combinations of the selected device architectures. If the design requires more than one device, the design is automatically partitioned across multiple devices. PLDesigner-XL also lets you choose among many speed, power, and package type variations offered by TI. Device selection design constraints include:

- Maximum number of devices
- Manufacturer
- Logic family
- Package type
- Temperature range
- Total price
- Maximum propagation delay
- Minimum frequency
- Maximum current usage (power)
- Total number of pins in design
- Two user constraints

To prioritize the constraints, you simply add a relative weighting. For example, price may be given a weight that is twice as important as power.

Two user constraint fields let you enter data for a device that is specific to your design or manufacturing environment. These two fields have the same priority capabilities as the standard constraint fields.

If manual device selection is preferred, you may specify the devices for implementation in your physical information file. This procedure will override the device selection process.

4.3.4.2 FPGAs

Device selection for FPGA devices is done with a Physical Information (`.pi`) file. This file specifies the FPGA device type and the design signals to be placed in the device. The `.pi` file can be used to specify any programmable device that PLDesigner-XL supports. The `.pi` file also gives the designer complete control over PLD, CPLD, and FPGA implementation. An example `.pi` file is listed below. This `.pi` file will direct PLDesigner-XL to partition the specified design pins into the TPC1280 FPGA device with the rest of the design pins being automatically partitioned over other TI PLD/CPLD devices.

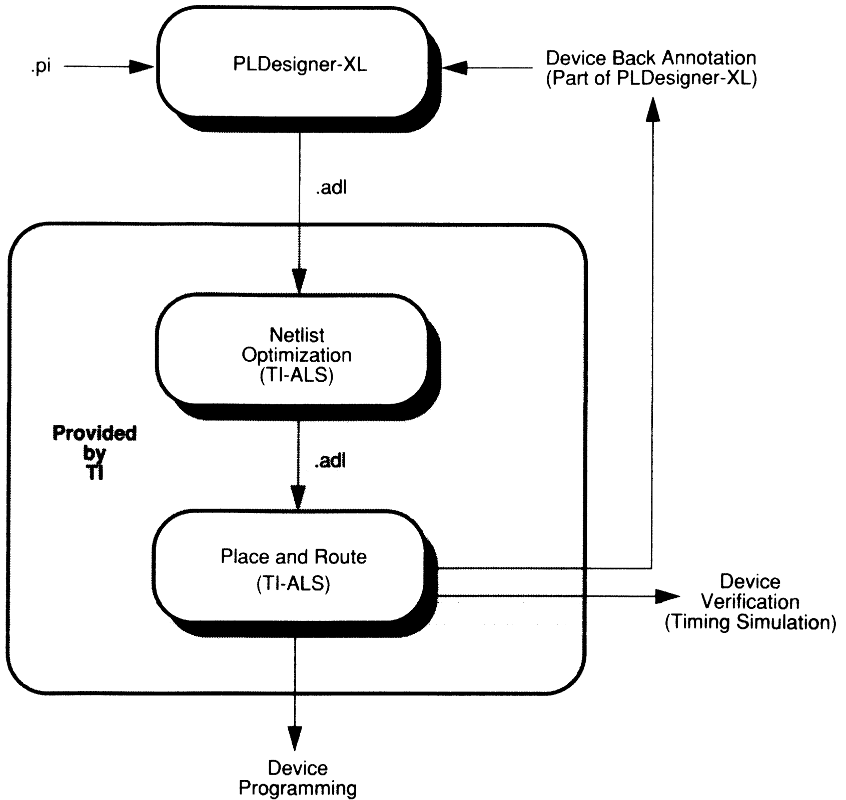
```
.pi file example :          fixed group
target 'PART_NUMBER TI TPC1280BG-177C' ;
out1:3 ;
out2:5 ;
out3:22;
end group;
```

4.3.4.3 FPGA Backend Software

To complete the FPGA device implementation, you need Place and Route software from TI. This software allows the user to implement, program, and (optionally) use a TI supported timing simulator to verify the device.

Figure 4-14 shows the flow when using PLDesigner-XL to target a TI FPGA for backend Place and Route. This method allows a user to synthesize a design description directly into a TI FPGA device(s).

Figure 4-14. Flow Diagram Using PLDesigner-XL to Target a TI FPGA



4.3.4.4 Back Annotation

Back annotation lets you send pin and package information for the targeted device back into PLDesigner-XL for documentation of the design. Once an FPGA design has been through Place and Route, PLDesigner-XL can read the TI Place and Route files and extract the FPGA package and pin information for documentation of the design. The interconnect between the TI FPGA and the other TI PLD/CPLD devices in the partitioned solution are also documented.

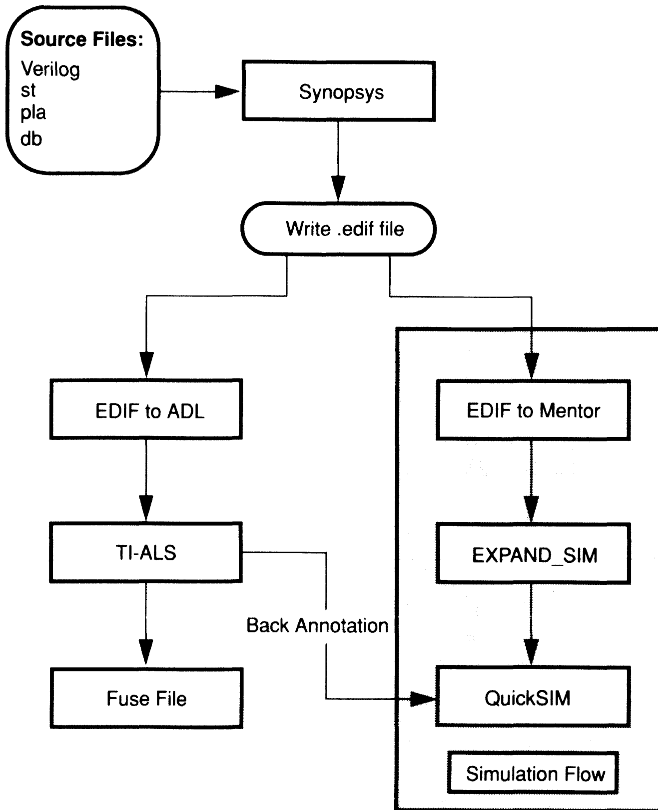
4.4 Synopsys to FPGA Migration†

4.4.1 Introduction

This section describes in detail the complete design flow from high-level circuit description to an FPGA and will help the beginner with the Synopsys and TI-ALS design tools. The high-level description language provides a technology-independent design source. Therefore, this flow shows the migration for designs from most technologies onto an FPGA. Figure 4-15 illustrates the general design flow.

† Contributed by Andrew Chapman, FPGA Applications, Texas Instruments Ltd.

Figure 4-15. Synopsys to FPGA Migration Flow



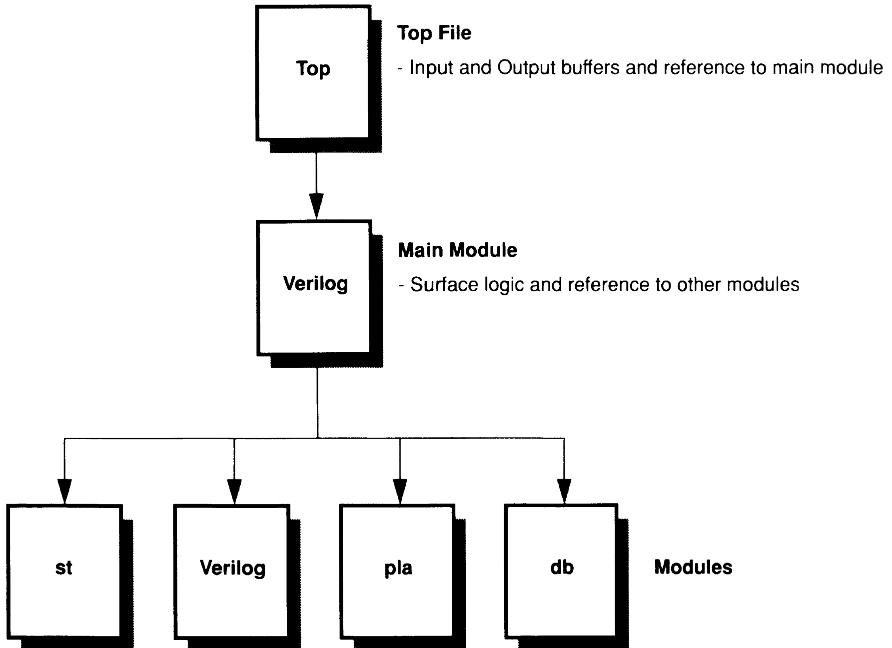
4.4.2 Synopsys to FPGA Flow

Synopsys optimizes input files to produce the correct speed and area considerations for the design. The final output form is an `.edif` file. The `.edif` file is converted into Mentor schematics and expanded to produce the simulation database. Evaluation of the design takes place using QuickSIM™. Once the design is fully functional, the `.edif` file (created in Synopsys) is converted into a fuse file using TI-ALS software. The postlayout delays can then be back annotated into QuickSIM to resimulate the design.

4.4.3 A Brief Overview of Synopsys

Synopsys is a circuit synthesis tool. It can take several forms of source files and produce a circuit according to a set of instructions. The program's goal is to meet all the constraints placed upon that particular circuit. It produces designs that best fit all of the specified requirements. To improve the speed of the circuit, the compiler compromises on the area restraint. Synopsys compiles the source files independently, but they need to be referenced in a main module. This main module contains references to the other modules and any surface logic and is referenced in a top file that contains the input and output buffers. A diagram of the source file hierarchy is illustrated in Figure 4-16.

Figure 4-16. Synopsys File Hierarchy



4.4.4 Compilation Strategy for Synopsys to FPGA Flow

The FPGAs are unique in timing and area requirements that have to be met when compiling the source files. Compilation methods for the various types of files were established by trial and error. The area of the design is an important consideration for the type of FPGA used. A module is the basic unit of the device that can be configured for any of the cells in the FPGA libraries. Module counts for the different FPGAs are listed in Table 4-1.

Table 4-1. Module Count for the Different FPGAs

Device	Module Count
TPC1010	295
TPC1020	547
TPC1225	430
TPC1240	684
TPC1280	1232

These areas are total module counts. Ensure that your design will fit into these areas with 5 to 10 logic modules remaining. Otherwise, the TI-ALS software may have difficulty routing the design.

4.4.5 Starting Synopsys

This design flow was completed within an AEGIS™ environment on an Apollo™ Domain™ network. The version of Synopsys used was 2.2(a); however, the Synopsys commands can be used under any Synopsys environment on any supporting system. The main program run by Synopsys is `dc_shell`. To run this, connect to the correct node containing the Synopsys software.

In an AEGIS environment, type

crp -on *node_name* -me

Change to a UNIX shell and type

/bin/csh

To run Synopsys area, type

dc_shell

While loading **dc_shell**, Synopsys runs a set-up .synopsys file. This must be present in the directory under use since it sets up the FPGA libraries correctly. Here is an example of the set-up .synopsys file:

```
search_path = { <location of libraries>/synopsys_2.2/
libraries/syn };

 link_library   = { TSC700_COM_MAX.db TPC10.db } ;
 target_library= { TPC10.db } ;
 symbol_library= { tpc10.sdb, mentor_specials.sdb } ;
 generic_symbol_library = "generic.sdb"
lib_name = "TPC10.db"
lib_file = "TPC10.db"
read lib_file
font_library = "1_25.font"
synthetic_library= {};
edifout_power_and_ground_representation = net
edifout_pretty_print = true
write_name_nets_same_as_ports = false
edifout_power_and_ground_to_interfaces = false
edifout_instantiate_ports  = true
edifin_lib_route_grid  = 60
edifout_ground_name=GND ;
edifout_power_name=VDD ;
edifout_ground_net_property_name=GLOBAL ;
edifout_ground_net_property_value=GND ;
edifout_power_net_property_name=GLOBAL ;
edifout_power_net_property_value=VDD ;
read_array_naming_style=%s_%d ;
compile_fix_multiple_port_nets=true ;
write_name_nets_same_as_ports=true ;
include alias.syn ;
```

To leave Synopsys at any time, type

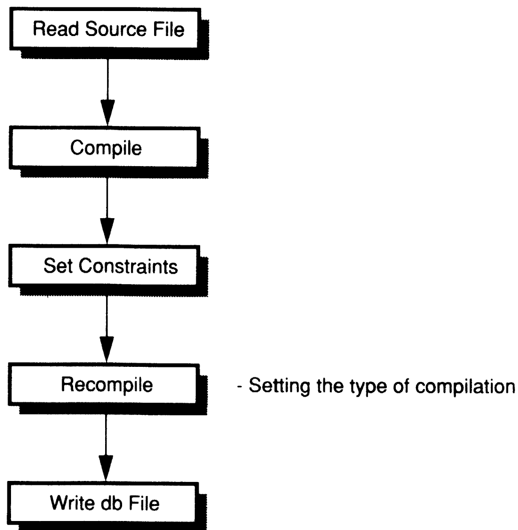
```
dc_shell> quit
```

To stop a process running in Synopsys, press CTRL Q once. This should halt the process; but, if it looks stuck, press CTRL Q again. If this does not work, press CTRL Q to exit from Synopsys.

4.4.6 Specific Compilation for Source Files

The Mentor path of this flow requires a schematic form of the .edif file to compare to the netllist form, for the TI-ALS path. This requires different approaches when using Synopsys (See Figure 4-17 for Synopsys compilation flow).

Figure 4-17. Synopsys Compilation Flow



The basic source files can be compiled in Synopsys using the following commands for each one.

- ❑ st (state table)

```
read -format st xxxx.st
compile
report_area
set_fsm_minimize true
max_fanout 6
max_delay 0.0 all_outputs()
clocks_at xx xx xx CLK
set_arrival  0.0   {"Inputs","...","...",etc.}
max_area 0.0
compile -incremental_mapping
write -format db -hierarchy -output xxxx.db
```

- pla (programmable logic array)

```
read -format pla xxxx.pla
check_design
compile
max_area 0.0
max_fanout 6
max_delay 0.0 all_outputs()
compile -incremental_mapping
write -format db -output xxxxxx.db
```

- verilog

```
read -format verilog xxxx.v
check_design
compile
max_area 0.0
max_fanout 6
max_delay 0.0 all_outputs()
clocks_at xx xx xx CLK
set_arrival  0.0   {"Inputs","...","...",etc.}
set_drive  -rise  drive_of  (  -rise  TPC10/AND2/Y  )
all_inputs()
set_drive  -fall  drive_of  (  -fall  TPC10/AND2/Y  )
all_inputs()
set_load load_of ( TPC10/BUFA/A) all_outputs()
compile -incremental_mapping
write -format db -output xxxxxx.db
report_clock -constraint
```

- db (database) Files

```
(You can just read db files.)
read -format db design_name.db
(You will still have to produce a top_file.)
```

Top files connect the input and output ports to the design. They do not need to be compiled. There are two compilation strategies, described below. The first creates a netlist `.edif` file for use with the TI-ALS software. The second process creates a schematic `.edif` file for Mentor, to simulate the design. To create an edif file for use in Mentor it is necessary to use a DUMMY file, shown in Figure 4-18.

Figure 4-18. DUMMY File

```
module S2M_BUSFIX_TOP;

   <new top design name>  I1 ();
   S2M_BUSFIX_CELL        I2 ();
endmodule

module S2M_BUSFIX_CELL (A,Y);
  input   [35:0] A  ;
  output  [35:0] Y  ;

  INV I0 (.A(A[0]), .Y(Y[0]));
  INV I1 (.A(A[1]), .Y(Y[1]));
  INV I2 (.A(A[2]), .Y(Y[2]));
  INV I3 (.A(A[3]), .Y(Y[3]));
  INV I4 (.A(A[4]), .Y(Y[4]));
  INV I5 (.A(A[5]), .Y(Y[5]));
  INV I6 (.A(A[6]), .Y(Y[6]));
  INV I7 (.A(A[7]), .Y(Y[7]));
  INV I8 (.A(A[8]), .Y(Y[8]));
  INV I9 (.A(A[9]), .Y(Y[9]));
  INV I10 (.A(A[10]), .Y(Y[10]));
  INV I11 (.A(A[11]), .Y(Y[11]));
  INV I12 (.A(A[12]), .Y(Y[12]));
  INV I13 (.A(A[13]), .Y(Y[13]));
  INV I14 (.A(A[14]), .Y(Y[14]));

  INV I15 (.A(A[15]), .Y(Y[15]));
  INV I16 (.A(A[16]), .Y(Y[16]));
  INV I17 (.A(A[17]), .Y(Y[17]));
  INV I18 (.A(A[18]), .Y(Y[18]));
  INV I19 (.A(A[19]), .Y(Y[19]));
  INV I20 (.A(A[20]), .Y(Y[20]));
  INV I21 (.A(A[21]), .Y(Y[21]));
  INV I22 (.A(A[22]), .Y(Y[22]));
  INV I23 (.A(A[23]), .Y(Y[23]));
  INV I24 (.A(A[24]), .Y(Y[24]));
  INV I25 (.A(A[25]), .Y(Y[25]));
```

```
    INV I26 (.A(A[26]), .Y(Y[26]));
    INV I27 (.A(A[27]), .Y(Y[27]));
    INV I28 (.A(A[28]), .Y(Y[28]));
    INV I29 (.A(A[29]), .Y(Y[29]));
    INV I30 (.A(A[30]), .Y(Y[30]));
    INV I31 (.A(A[31]), .Y(Y[31]));
    INV I32 (.A(A[32]), .Y(Y[32]));
    INV I33 (.A(A[33]), .Y(Y[33]));
    INV I34 (.A(A[34]), .Y(Y[34]));
    INV I35 (.A(A[35]), .Y(Y[35]));

endmodule
<Copy ALL of the top file and place the new top file here.>
```

The ESIread™ software used in the EDIF to Mentor conversion script looks at the first bus in the hierarchy and thinks that it is looking at the largest bus in the design. This means that large buses after a short one are cut to the smaller size. This can cause major problems. To overcome this, a DUMMY file is added over the top of the design. This has a 32-bit bus that fools the software into reading the design properly. This DUMMY file specifies the design top file and this large bus. The script removes the large bus and continues to process the design.

The top file and DUMMY files are compiled using the following commands:

- Top File

```
read -format verilog <design name>_top.v

link

translate

write -format db -hierarchy -output fpga1.db

include set_edifout_als.syn

include synopsys_setup_tgc1000_tpc10

edifout_netlist_only = true

write -format edif -hierarchy -output design_top.edif
```

- DUMMY

```
read -format verilog DUMMY
link
translate
include set_edifout_als.syn
include synopsys_setup_tgc1000_tpc10
edifout_netlist_only = false
create_schematic -size mentor_maximum -hierarchy
write -format edif -hierarchy -output
design_mentor.edif
```

There are several include scripts which configure the compilation of the above files. These are explained and listed in Figure 4-19.

Initialize the circuit for TI-ALS conversion by typing

```
set_edifout_als.syn
```

Figure 4-19. Include Scripts Which Configure the Compilation of the Top and DUMMY Files

```
edifout_netlist_only = "true"
edifout_external = "false"
edifout_no_array = "true"
edifout_power_and_ground_representation = "net"
edifout_ground_name = "GND"
edifout_ground_net_property_name  = "GLOBAL"
edifout_ground_net_property_value = "GND"
edifout_power_name = "VDD"
edifout_power_net_property_name  = "GLOBAL"
edifout_power_net_property_value = "VDD"
edifin_ground_net_property_name  = "GLOBAL"
edifin_ground_net_property_value = "GND"
edifin_power_net_property_name   = "GLOBAL"
edifin_power_net_property_value  = "VDD"
edifout_ground_pin_name = "DOES_NOT_MATTER"
edifout_power_pin_name  = "DOES_NOT_MATTER"

synopsys_setup_tgc100_tpc10
```

Figure 4-20 shows an include file which sets up the correct library for creating the schematics.

Figure 4-20. Include File That Sets Up Correct Library for Creating the Schematics

```
edifout_array_member_naming_style =  "%s(%d)"
edifout_array_range_naming_style  =  "%s(%d:%d)"
edifout_external                  =  "true"
edifout_ground_name               =  "gnd"
edifout_ground_net_property_name  =  "gnd"
edifout_ground_net_property_value =  "0SF"
edifout_ground_pin_name           = "gnd"
edifout_instance_property_name    = ""
edifout_instantiate_ports         = "true"
edifout_merge_libraries           = "false"
edifout_netlist_only              = "false"
edifout_no_array                  = "false"
edifout_pin_direction_in_value    = "in"
edifout_pin_direction_inout_value = "inout"
edifout_pin_direction_out_value   = "out"
edifout_pin_direction_property_name = "PINTYPE"
edifout_pin_name_property_name      = ""
edifout_portInstance_disabled_property_name  = ""
edifout_portInstance_disabled_property_value = ""

edifout_portInstance_property_name           = ""
edifout_power_and_ground_representation = "net"
edifout_power_name                      = "vcc"
edifout_power_net_property_name         = "vcc"
edifout_power_net_property_value        = "1SF"
edifout_power_pin_name                  = "vcc"
edifout_skip_port_implementations       = "false"
edifout_target_system                   = "mentor"
edifout_translate_origin                = "center"
edifout_pretty_print                    = "true"
search_path = search_path + .
search_path = search_path + /user/daveb/synopsys/tgc1000_4.0/binary
search_path = search_path + /user/madams/fpga/libraries/synopsys/a1200
symbol_library = {tpc10.sdb mentor_specials.sdb}
link_library = {TPC10.db TPC10S.db}
target_library = {TPC10.db TPC10S.db}
link_path = {tgc1000_2_tpc10.db, TGC1000_COM_MAX.db, TPC10.db TPC10S.db}
```

4.4.7 Other Useful Tips

Buses in a design must be of the format [0:27], not [27:0]. Otherwise buses may be inverted when the `.edif` file is converted, which can cause huge problems. To avoid this problem, just remove all the buses before you start compiling. The current Mentor conversion script cannot handle buses; if you do have to use buses in the design, rip them out just after you have read in the source files by typing the following command:

remove_bus ***bus_name***

If your design has any tie-off to GND or VDD, some precautions have to be taken in order to compile the circuit properly. In the Verilog source files the tie-off cells are written like this,

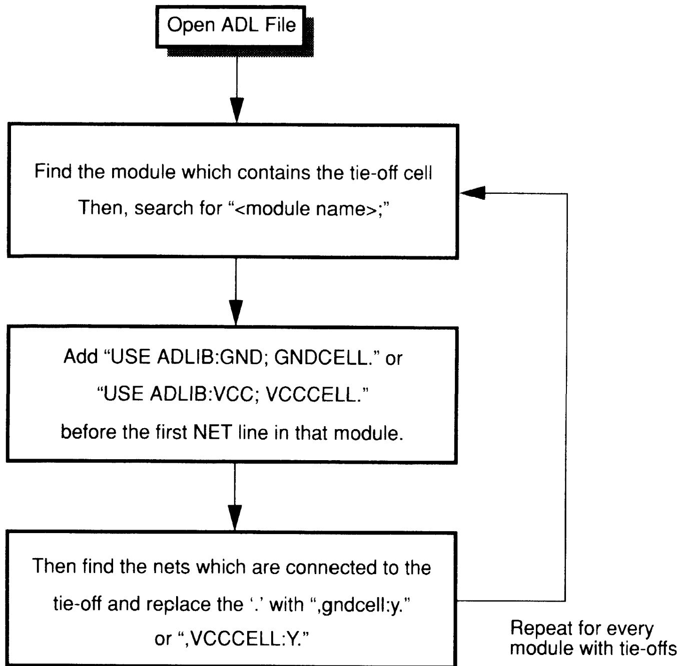
```
GND inst<instanst number> ( .Y(<net name>) );
VCC inst<instanst number> ( .T(<net name>) );
```

The TI-ALS software has problems with GND and VCC connections. To solve this, you have to edit the `.adl` file produced by the TI-ALS software later in the design flow. The ADL correction flow is illustrated in Figure 4-21.

Once the corrections have been done, you must certify the `.adl` file to produce the correct checksum by typing the following:

certify adl ***design_name***

Figure 4-21. ADL Correction Flow



4.4.8 EDIF to Mentor Conversion

This script takes the `.edif` file created in Synopsys and produces Mentor files with schematics. It also removes the DUMMY cell added to make sure that **ESIread** works properly.

To run the conversion script, type

crp -on *node_name* -me

To change to a UNIX shell, type

/bin/csh

Then run the following script:

s2m_edif *edif_file_name directory_for_designs*

fpga_library name_of_top_design

For a 500-module design, this process takes 10 minutes. While it is running, the errors shown in Figure 4-22 will occur and are acceptable; ignore them.

Figure 4-22. Conversion Errors

```
# ?    Error: symbol could not be opened. (from ESI/ESIread 2)
# ?    Error: could not add property 'NET'. (from ESI/ESIread 23)
# ?    Warning: Net name "NCLK" connects the following vertices:
# ?            V$3807     V$3801     (From Idea/NetCheck/Net Name 03)
```

4.4.9 Mentor to QuickSIM

To produce the simulation files, you must type the following:

expand_sim *directory_name_for_mentor_files*

Once this has been completed, type the following command to run QuickSIM.

quicksim *directory_name_for_mentor_files*

4.4.10 Running TI_NETED

To run this program, type

source tide_info

To set up the links for **ti_neted**, type

ti_neted *directory_name_of_design*

To do hand edits to the schematics you must load the library by typing:

read menu *directory_of_ALS symbols*/als/lib/a1000/a1000

This loads the `a1000` libraries in the component libraries menu. You can now add any of the components in the FPGA library. When checking each sheet press SHIFT and the SAVE/EDIT key; this process removes the TI check parameters and saves time. To move up and down sheets, first select them and then use the FILE/SHOW SHEET/DOWN FOR EDIT option.

4.4.11 Running the TI-ALS Software

The TI-ALS software has three main stages.

- Converting `.edif` files to `.adl` files
- Laying out the FPGA (which runs five programs)
- Checking the postlayout timing of the circuit.

To run the EDIF to ADL converter, first connect to the software node with this command:

crp -on *node* -me

To create the link to the software, execute the following:

csr -a /user/als/bin

4.4.12 Run Conversion Program

To run the conversion program, type:

edn2adl fam:act1 ednin:*edif_file* edninflavor:synps *top_design*

This program makes a directory and creates some files underneath. If you have tie-offs in your circuit, the `.adl` file will have to be changed at this point. If you want to set any or all of the pins, see the ADL conversion flow for specific nets. On the FPGA, the `.ipf` file will have to be altered.

The TI-ALS software automatically places nets to pins; this can prove to be very awkward for PCB design. To prevent this, you can fix the position of some or all of the pins. This is done by editing the `.ipf` file that is created when changing EDIF to ADL. After the header section, write the following commands.

The header starts with

```
; header
```

and finishes with

```
; endheader
```

Type:

def *design_name*

Then write

```
net < i/o pin name >/;;
pin:< pin number >,
fix:''.
etc.
```

and finish with

```
end.
```

The **fix:' '.** command fixes the position of the pin. If you miss any of the pins, the TI-ALS software will place them for you. All TI-ALS files have a checksum, so the `.ipf` file must be certified again; this is done by typing

certify ipf *design_name*

Once this is all done, the FPGA layout can be produced. This is done by running several programs in succession. If any one fails, then the whole layout placement will fail. The programs are as follows:

- Validate

  Checks the design for area and fan-out errors.

- Ioplace

  Automatically place the pins or reads your `.ipf` file if you have changed it.

- Place

  Places the designs on the FPGA.

- Route

  Connects up the logic modules.

- extract.als

  Extracts the `.als` files.

- Fuse

  Produces the fuse file needed to program the FPGAs.

All these programs can be run in a user-defined script such as **layout_fpga** shown in Figure 4-23. Run this script by typing the following command with the correct options.

```
layout_fpga  [family:fam] top_design
             [die:device]
             [package:pkg]
```

Where valid series values are

fam	**1000, 1200**
device	**1020, 1240, 1280**
pkg	**lcc68, lcc84, pga84, pga176, pga132,qfp144, qfp160, qfp100, qfp84**

Figure 4-23. Layout_fpga Script

```
eon
alsbin := "//node_2d99c/user/als/bin"
if eqs ^1 '' then
   args "Usage: layout_fpga FAMILY:ffff DIE:dddd PACKAGE:pppppp <design-
                  name>"
  args "       Where ffff   = 1000 or 1200,"
  args "             dddd   = 1020 or 1280,"
  args "             pppppp = lcc44, lcc68, lcc84, pga84, pga176, pga132,
                  qfp144, qfp160, qfp100, qfp84."
  exit
endif
^alsbin/validate    ^1 ^2 ^3 ^4
^alsbin/ioplace     ^1 ^2 ^3 ^4
^alsbin/place       ^1 ^2 ^3 ^4
^alsbin/route       ^1 ^2 ^3 ^4
^alsbin/extract.als ^1 ^2 ^3 ^4
^alsbin/fuse        ^1 ^2 ^3 ^4
eoff
```

All the errors and warnings must be resolved before an error-free fuse file can be produced. Most of the errors concern the large fan-outs. All the fan-outs should be lower than 10, especially clock signals. TI-ALS tolerates a maximum fan-out of 24, but this is *not* recommended. Any fan-outs over 24 will cause an error and stop the process. To take a closer look at the postlayout timing, run Timer.

4.4.13 Timer

Run this software after **layout_fpga** by typing:

```
timer        [temp:temp]                  design_name
             [volt:voltage_range]
             [speed:speed_grade]
             [proc:process]
             [timing:timing]
             [die:device]
             [package:pkg]
             [design:design_name]
```

Where valid series values are

temp	-55, -40, 0, 25, 70, 85, 125
voltage_range	4.5, 4.75, 5.0, 5.25, 5.5
speed_grade	-1, -2, standard
process	worst, best
timing	post, pre
device	1020, 1240, 1280
pkg	lcc68, lcc84, pga84, pga176, pga 132, pga144, qfp160, qfp100, qfp84

For further explanation on how to use *Timer*, consult the *FPGA Software Reference Manual.*

4.4.14 Back Annotation

It is possible to test the postlayout timing of an FPGA in QuickSIM. This can be done by running **export.als** on a `.del` file created using **layout_fpga**. This changes the Mentor database to include the postlayout delays. The commands for this are as follows. First, copy the `design.del` file from the `\als` directory to your Mentor directory.

Run conversion program shown in Figure 4-24.

Figure 4-24. Conversion Program

```
export.als    [tempr:temperature_range]
              [voltr:voltage_range]
              [speed_grade:speed_grade] design_name
```

Where valid series values are

```
temperature_range ind, com, mil
voltage_range     ind, com, mil
speed_grade       std, -1, -2
```

This has now altered the Mentor files so that they have postlayout characteristics. The test description language (TDL) can now be run, as before, on QuickSIM.

4.4.15 Producing an FPGA

The first operation to be performed is to copy the fuse file to an IBM PC so that you can use the Activator programming unit.

```
c:\ cd designs
```

Make a subdirectory using the name of your design as follows:

```
c:\designs\md design_name
```

Then move down to this directory and copy all the TI-ALS files from a floppy disk. For example,

```
c:\designs\cd design_name
```

```
c:\designs\design_name\copy a:\design_name.*
```

Once this is done, return to the root directory by invoking:

```
c:cd\
```

Run the TI-ALS software by typing

```
c:\als design_name
```

This starts the TI-ALS software. After the software has loaded, a menu screen appears. First, specify the design and device to be blown. Move the cursor onto the menu column, select Project and press RETURN. This produces another short menu with Design and Device options. Check that the design name is the correct one, then choose the device option. The screen displays the following device choices: 1010, 1010A, 1020, and 1020A. For a 68-pin PLCC chip, the device is 1020A. Once this is decided, you must choose the type of package you are using.

4.4.16 Activator

Now you are ready to program the device. Place the device in the Activator, with pin 1 facing you. Close the door. Choose *Activate* on the menu screen. This initializes the Activator and brings up a new menu screen. Ensure that the device is blank; to do this, select *BlankCHK* on the menu screen. When the device passes, it can be programmed by selecting *Program* on the menu. This process takes from 2 to 20 minutes, depending on the size of your design. The Activate program may bring up warnings about files concerning the last update time; these can be ignored. A successful program message displays when the operation is completed.

The next menu selection allows the Program fuse to be blown (and prevents the FPGA from being programmed again). The Probe fuse can also be blown, but this is not recommended because it prevents the use of the Actionprobe to debug the circuit. The FPGA can now be removed from the Activator and plugged into a circuit. Select *Exit* on the menu column to return to the main screen. Select *Exit* again to quit the TI-ALS software.

4.4.17 Actionprobe

This piece of equipment allows you to look at any two internal nets in the FPGA while it is plugged into its circuit. You can also provide stimulus for the circuit. To run the Actionprobe, choose *Debug* on the main menu. Place the Actionprobe in the circuit socket, put the FPGA in the top and push it down. Select *icp* to assign internal nets to the probe pins. Hierarchical nets can be accessed by quoting the module name, for example,

```
/core/fend/inst12;q
```

The `.adl` file lists all the nets, and is a useful reference when locating internal nets. The internal nets can then be monitored via pins 58 and 59, with an oscilloscope or logic analyzer.

4.4.18 Summary

This is a detailed look at producing circuits on an FPGA using a high-level description language. This means that designs can be prototyped in an FPGA within the time span of a few days. Once a design has been proven, the same high-level description language files can be used to produce the circuit on another media, like an ASIC.

4.5 12-Hour Clock—A High-Level Design Flow Example†

4.5.1 Introduction

This section outlines the high-level design flow used to implement a 12-hour digital clock. The design is described in Verilog HDL, synthesized to a netlist description using Synopsys, and targeted to a TPC10 series FPGA using the Texas Instruments Action Logic System (TI-ALS). The design is simulated using the Cadence VerilogXL™ simulator. This document assumes that the user is familiar with simple Verilog HDL, but if you need more information. please see Cadence Verilog manuals and tutorials.

4.5.2 Design Overview

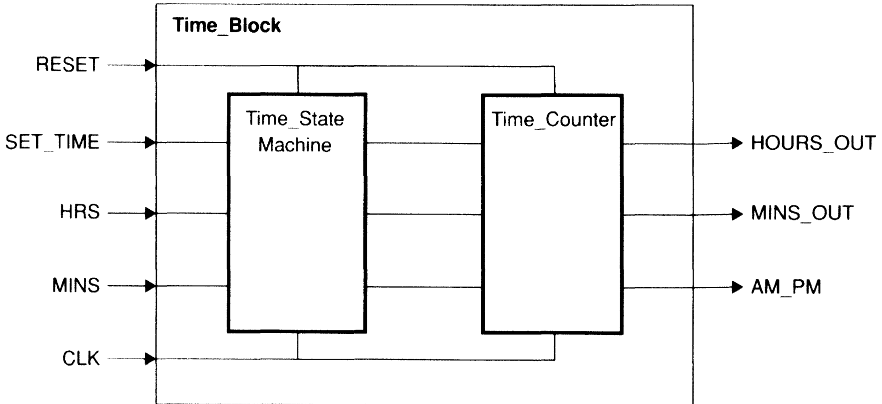
There are five inputs to the top-level block:

- RESET
- SET_TIME
- HRS
- MINS
- CLK.

SET_TIME is used in conjunction with HRS or MINS to set the time of day and CLK is the system clock. There are three outputs from the design: a 4-bit HOURS_OUT bus, a 6-bit MINUTES_OUT bus, and an AM-PM signal. The design contains two levels of hierarchy: TIME_STATE_MACHINE and TIME_COUNTER; both are called from the top-level block: TIME_BLOCK. These blocks and the hierarchy are illustrated in Figure 4-25.

† Contributed by Katey Derbyshire, FPGA Applications, Texas Instruments Ltd.

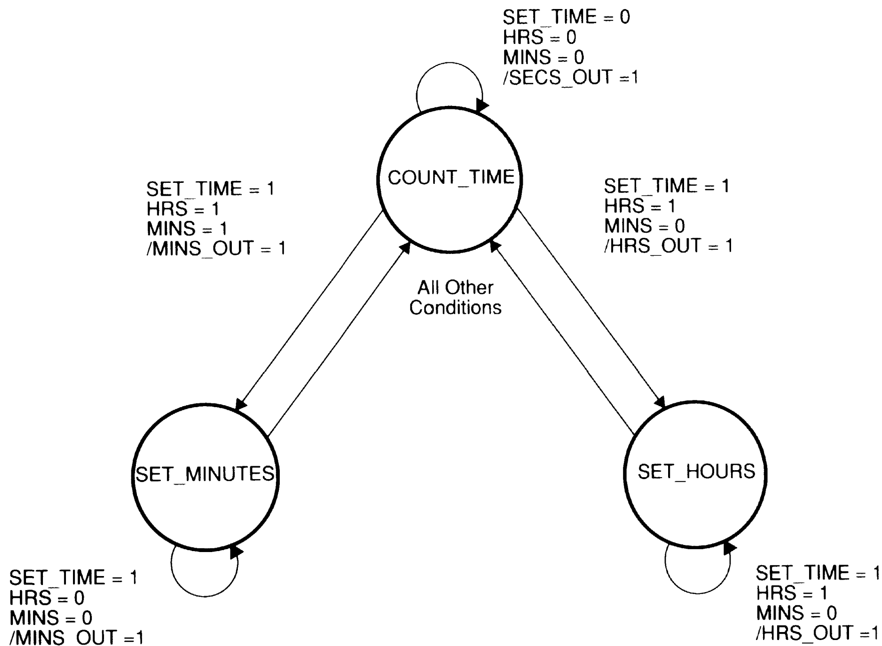
Figure 4-25. Block Diagram



- The TIME_STATE_MACHINE block is used to set the time of day and has three states whose relation is shown in Figure 4-26. While in the COUNT_TIME state, a pulse is output every second to the TIME_COUNTER block. The machine will remain in this state until SET_TIME and HRS or MINS are set. In SET_HOURS, a MINS_OUT pulse is fed to the TIME_COUNTER and minutes are incremented. In the same way, SET_HOURS generates a HRS_OUT pulse which then increments the hours counter.

- The TIME_COUNTER block increments the hours and minutes and sets the AM-PM output.

Figure 4-26. Timer State Diagram



Part of the Verilog HDL implementing this state machine is shown in Figure 4-27. A case statement is used to describe the three states. The remaining HDL code for the complete design can be found in the Synopsys tutorial directory which accompanies release 2.2b of the software.

Figure 4-27. Time_state_machine

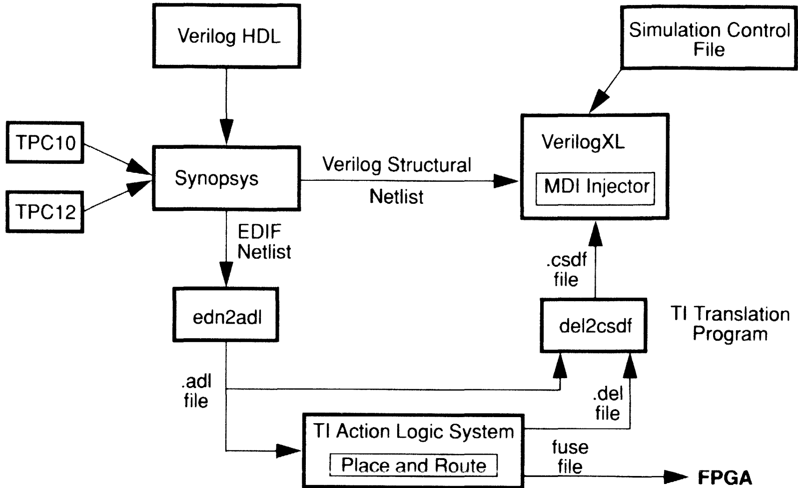
```
module TIME_STATE_MACHINE (RESET, TIME_BUTTON, HOURS_BUTTON,
MINUTES_BUTTON), input RESET, TIME_BUTTON, HOURS_BUTTON, MINUTES_BUTTON,
CLK;
output SECS, HOURS, MINS;

parameter COUNT_TIME=0, SET_HOURS=1, SET_MINUTES=2;

reg [1:0] CURRENT STATE, NEXT_STATE;
reg SECS, HOURS, MINS;

always @ (CURRENT_STATE or TIME_BUTTON or HOURS_BUTTON or MINUTES_BUTTON)
begin
      SECS = 0
      HOURS = 0
      MINS = 0;
      NEXT_STATE = CURRENT STATE;

      case (CURRENT_STATE) //synopsys full_case parallel_case

      COUNT_TIME: begin
            if (TIME_BUTTON & HOURS_BUTTON & !MINUTES_BUTTON)
                  begin
                  NEXT_STATE = SET_HOURS;
                  HOURS = 1;
                  end
            else if (TIME_BUTTON & HOURS_BUTTON & !MINUTES_BUTTON)
                  begin
                  NEXT_STATE = SET_MINUTES;
                  MINS = 1;
                  end
            else
                  begin
                  NEXT_STATE = COUNT_TIME;
                  SECS = 1;
                  end
            end
```

```
        SET_HOURS: begin
                if (TIME_BUTTON & HOURS_BUTTON & !MINUTES_BUTTON)
                        begin
                        NEXT_STATE = SET_HOURS;
                        HOURS = 1;
                        end
                else
                        begin
                        NEXT_STATE = COUNT_TIME;
                        SECS = 1;
                        end
                end
        SET_MINUTES: begin
                if (TIME_BUTTON & !HOURS_BUTTON & MINUTES_BUTTON)
                        begin
                        NEXT_STATE = SET_MINUTES;
                        MINS = 1;
                        end
                else
                        begin
                        NEXT_STATE = COUNT_TIME;
                        SECS = 1;
                        end
                end
        endcase
end

always @ (posedge CLK)
begin
        if (RESET)
                CURRENT_STATE = 2'b0;
        else
                CURRENT_STATE = NEXT_STATE;
end

endmodule
```

4.5.3 High-Level Design Flow

Figure 4-28 illustrates the high-level design flow used to take the Verilog HDL description to an FPGA fuse file.

Figure 4-28. Design Flow



The behavioral level Verilog is simulated using the module shown in Figure 4-29. This module calls up the top-level TIMER block and applies stimulus to the inputs. The simulator graphic environment is also set up within this module.

Figure 4-29. Test_timer.v File

```
module TEST;
reg RESET, SET_TIME, HRS, MINS, CLK ;
wire [3:0] CONNECT6;
wire [5:0] CONNECT7;
wire CONNECT8;

TIME_BLOCK TIME (RESET, SET_TIME, HRS, MINS, CLK, CONNECT6[3:0],
CONNECT7[5:0]

// Set up clock

always
begin
      $gr_waves ("SET_TIME%b", SET_TIME, "HRS....%b", HRS,
      "MINS...%b",MINS,
      "RESET..%b", RESET,
      "HRS_OUT%d", CONNECT6[3:0], "MINSOUT%d", CONNECT7[5:0], "AM_PM%"

      // Reset the Timer

      RESET = 1;
      SET_TIME = 0;
      HRS = 0;
      MINS = 0;
      #15 RESET = 0;

      // Set Timer to 11:30 AM in order to see PM transition

      // Set Hours = 11

      SET_TIME = 1;
      HRS = 1;
      MINS = 0;
      #110

      // Set Minutes = 30

      HRS = 0;
      MINS = 1;
      #400

      // Run clock until AM-PM Transition

      MINS = 0;
```

```
        SET_TIME = 0;
        #24000 $stop;
end

endmodule
```

Invoke the Verilog simulator with the following command:

ti_verilog time_block.v time_counter.v time_state_machine.v test_timer.v -f timer.opts

TI_Verilog is the Verilog executable compiled with the MDI Injector for insertion of postrouting delays. The `timer.opts` file contains library search paths, license numbers, and other simulation switches. An example of such an options file is shown in Figure 4-30.

Figure 4-30. Verilog Options File

```
-v /user/other/verilog/act1/libs/mylib
-y /user/other/verilog/act1/src/udps +libext+.v
-pxxxxxxxx
-pxxxxxxxx
+maxdelays
+define+prelayout
```

The Verilog graphic output is invoked from the test module using the **$gr_waves** command. The design's functionality can be verified here or further simulation can be performed from the Verilog prompt.

4.5.4 Design Synthesis

The behavioral-level Verilog is synthesized to a gate level description using the Synopsys design compiler. To use the TPC10 or TPC12 series library, the environment is set up in the `.synopsys` file. An example `.synopsys` file for TPC10 series is shown in Figure 4-31. The file includes a number of EDIF variables which must be set in order to generate an EDIF netlist which can be ported to the TI-ALS with the **edn2adl** command. More details on these variables can be found in the FPGA Synopsys documentation.

Figure 4-31. Example `.synopsys` *File*

```
 /* Search Paths */
search_path = { ./ <tpclib> [<user_defined_path>] };
link_path = {TPC10.db};
target_library = {TPC10.db};
symbol_library = {TPC10.sdb};
/* EDIF Variables */
              edifout_ground_name = "GND" ;
              edifout_ground_pin_name = "Y" ;
              edifout_netlist_only = "true" ;
              edifout_no_array = "true" ;
              edifout_power_and_ground_representation =
"cell";
              edifout_power_name = "VCC" ;
              edifout_power_pin_name = "Y" ;
              edifout_pretty_print = "true" ;
```

The synthesis process can be automated by use of a script file, an example of which is shown in Figure 4-32. Constraints are set to give the fastest implementation with realistic estimates for input drive strengths, output loads, and maximum fan-out.

Figure 4-32. Synopsys Control Script for Timer

```
/**********************************************/
/*    Synopsys control script for TIMER design   */
/*                                              */
/**********************************************/

free -all

/* Read HDL files */

read -format verilog CORE.v
read -format verilog COUNTER.v
read -format verilog STATE.v

/* Constrain design before compile */

current_design = CORE
update_timing
remove_constraint -all
max_delay 0 all_outputs()
max_period 0 all_clocks()
set_max_fanout 6.0 "CORE"
set_drive drive_of (TPC10/AND2/Y) all_inputs()
set_load load_of (TPC10/BUFA/A) all_outputs()

/* Compile the design */

compile -map_effort medium -verify -verify_effort low
create_schematic -size infinite -no_schematic -no_symbol_view
                -no_hierarchy_view -gen_database

/* Read I/O collar and write out verilog & edif netlists */

read -format verilog TIMER.v
create_schematic -size infinite -no schematic -no_symbol_view
                -no_hierarchy_view -gen_database
write -format verilog -hierarchy -output "TIMER_COMP.v"
write -format edif -hierarchy -output "TIMER.edif"
```

The design is synthesized with no I/O pads present; these are added at a later stage when the desired gate-level representation has been achieved. The I/O collar is read into the design compiler and the complete design is written as a Verilog netlist. No further compilation takes place at this stage. This I/O collar can be clearly seen on the top-level synthesis schematic shown in Figure 4-33.

Figure 4-33. Top-Level Synthesis Schematic

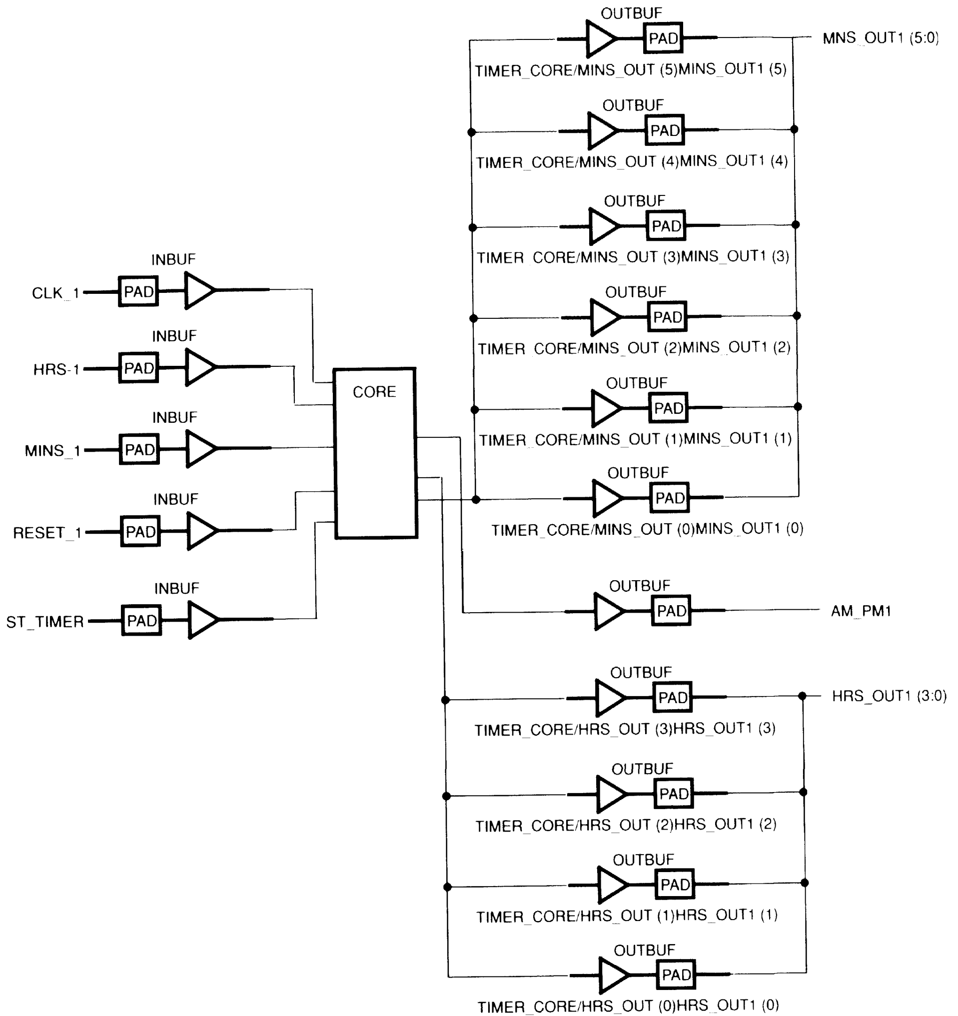
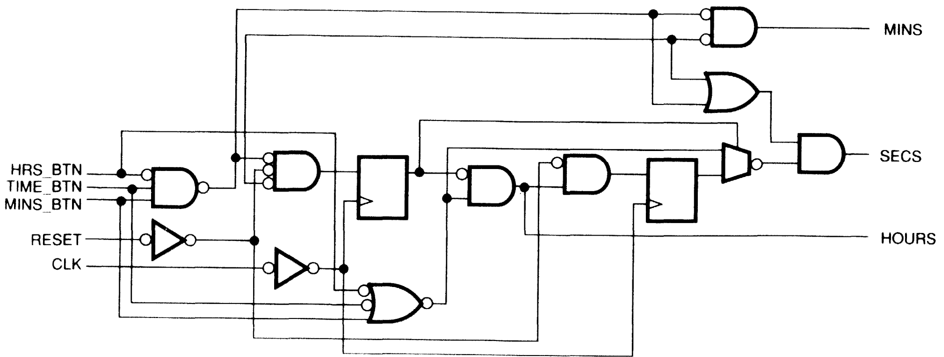


Figure 4-34 shows the gate-level implementation of the state machine after synthesis. TPC10 series library-specific macros, such as NAND3A, can be identified on the schematic.

Figure 4-34. TPC10 Series Implementation of TIME_STATE_MACHINE



4.5.5 TI-ALS

The TI-ALS requires an ADL netlist description which is obtained from the EDIF file generated by Synopsys (refer to the last line of the Synopsys control script in Figure 4-32), and uses the **edn2adl** tool provided with the TI-ALS installation as follows:

Usage: edn2adl fam:*value* ednin:*edif_netlist_filename*
[edninc fg:*configuration_filename*]
[edninflavor:*edninflavor*] *design_name*

Where:
Legal values for fam are **act1** and **act2**
Legal values for edninflavor are **dcs, wv, valid, synps**

For this application example, the command is

```
edn2adl fam:act1 ednin:TIMER.edif edninflavor:synps
timer
```

The `timer.adl` file is created; this `.adl` file is the netlist description of the design which interfaces to the TI-ALS. The design can now be ported to the TI-ALS environment.

The TI-ALS software is invoked by typing **als timer**. A TPC1010 device is selected in a 44 PLCC package. The design is validated and gives an initial module count of 160, with 16 I/O. Pins are assigned, the design is placed and routed, and physical timing information is then extracted from TI-ALS for

back annotation to Verilog. The TI-ALS is described in more detail in Chapter 3, *TI Action Logic System*.

4.5.6 Simulation

In order to simulate the structural Verilog netlist, several changes must be made to the test module described earlier. A **$monitor** statement is added and the output piped to one of two files, depending on whether the simulation is prelayout or postlayout. More importantly, a conditional block is added to call the MDI Injector when postlayout simulations are run. For the Timer, this block takes the form shown in Figure 4-35.

Figure 4-35. Conditional Statement to Invoke MDI Injector for Postlayout Simulations

```
'if def prelayout
'else
initial
begin
$TI_MDI_injector ("-scope", "TEST_TIMER_COMP.TIMER1", "-
d",
"TIMER.csdf", "-v");
end
'endif
```

The TI-ALS delay extraction produces a `.del` file containing all the net and component delays. The `.del` file is used in conjunction with the ADL netlist by the TI translator, **del2csdf**. This translator processes the delay information and generates a Cadence standard delay format (`.csdf`) file which is injected into VerilogXL using the MDI Injector routine.

The delays extracted from TI-ALS for typical operating conditions are

- 5V
- 25 degrees C
- Worst-case process
- Standard speed

The translator then scales these delays for minimum and maximum. All I/O buffers are assigned a constant delay, independent of routing. The TI translator, **del2csdf**, is called from the design directory as follows:

Usage: del2csdf *prefix_name_of_input_files*

For example,

del2csdf timer

The translator automatically inserts a timescale directive into the top-level design file, in this case, `timer.v`, which contains all the input and output buffers. The translator generates the output file, `timer.csdf`, which is then injected into VerilogXL for postlayout simulation.

Figure 4-36 shows the Verilog graphic output obtained from a prelayout simulation. After RESET goes away, SET_TIME and HRS are used to set HOURS_OUT to 11; SET_TIME and MINS are used to set MINS_OUT to 55 and then the clock is left running until an AM-PM transition is seen.

In order to compare the prelayout and postlayout simulations, a specific transition will be looked at more closely. The delay from CLK to a settled value on HOURS_OUT is 3 ns in the prelayout simulation. This delta time can be measured on the graphic display, shown in Figure 4-37 or can be deduced from the textual output obtained from the **$monitor** commands used in the test module shown in Figure 4-38.

Figure 4-36. Prelayout Simulation Result

Header: Prelayout TIMER		
User: FPGA group account		
Date: Aug 27, 1992 17:25:41	Time Scale From: 0 to: 1332000	Page: 1 of 1

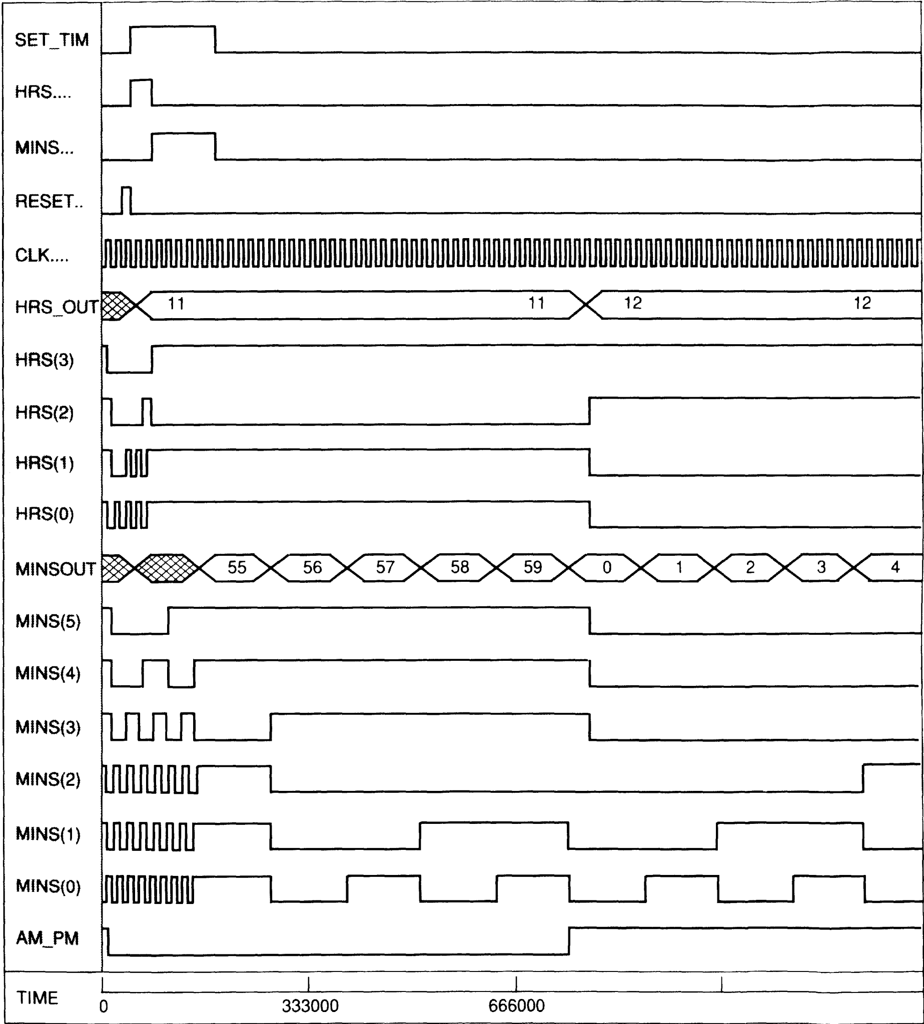


Figure 4-37. Prelayout Simulation, CLK to HOURS-OUT Delay of 3 ns

Header: Prelayout Timing		
User: FPGA group account		
Date: Sep 20, 1992 13:14:31	Time Scale From:11000 to: 15000	Page: 1 of 1

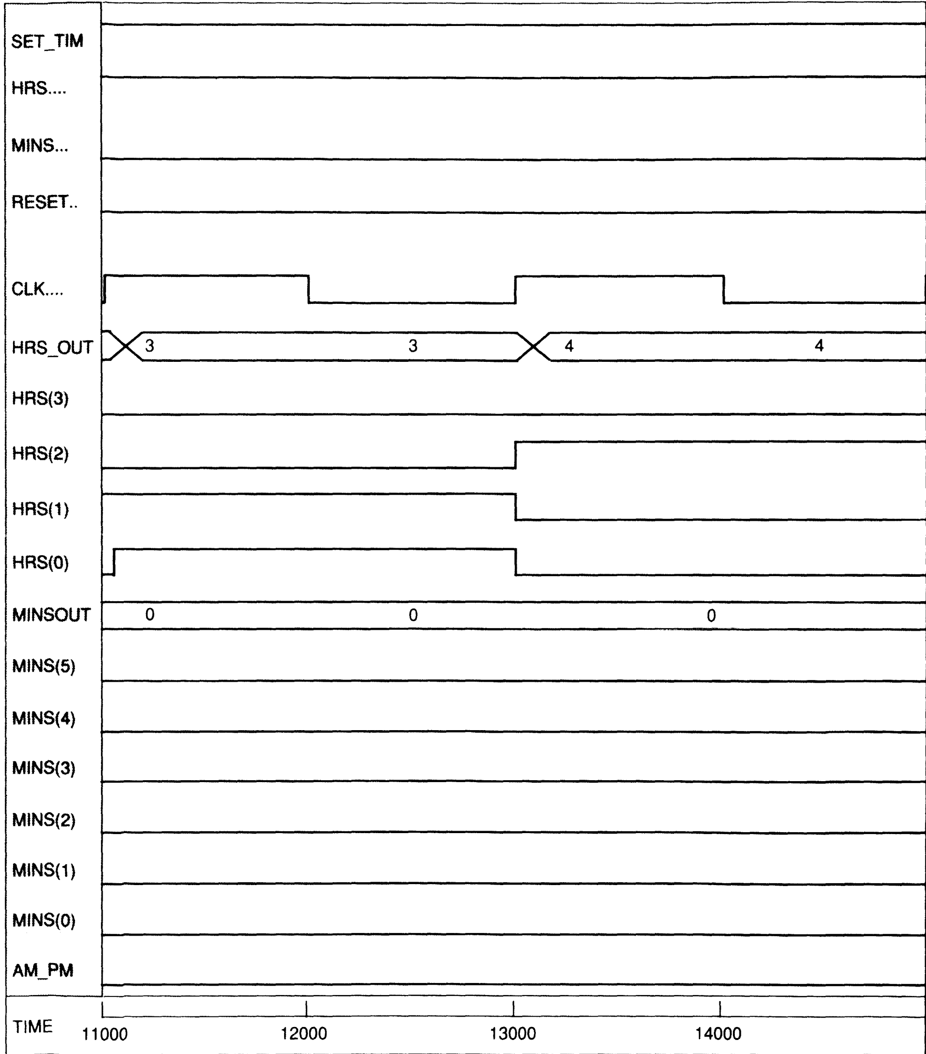


Figure 4-38. $Monitor Output

```
   0 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 0 CLK : 0 HRS_OUT : xxxx MINS_OUT : xxxxxx AMPM :x
 100 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : xxxx MINS_OUT : xxxxxx AMPM :x
 200 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 1 CLK : 0 HRS_OUT : xxxx MINS_OUT : xxxxxx AMPM :x
 300 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 1 CLK : 1 HRS_OUT : xxxx MINS_OUT : xxxxxx AMPM :x
 302 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 1 CLK : 1 HRS_OUT : 0xxx MINS_OUT : xx0xx0 AMPM :0
 303 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 1 CLK : 1 HRS_OUT : 0000 MINS_OUT : 000000 AMPM :0
 400 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 1 CLK : 0 HRS_OUT : 0000 MINS_OUT : 000000 AMPM :0
 500 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 1 CLK : 1 HRS_OUT : 0000 MINS_OUT : 000000 AMPM :0
 600 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 0 HRS_OUT : 0000 MINS_OUT : 000000 AMPM :0
 700 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0000 MINS_OUT : 000000 AMPM :0
 703 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0001 MINS_OUT : 000000 AMPM :0
 800 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 0 HRS_OUT : 0001 MINS_OUT : 000000 AMPM :0
 900 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0001 MINS_OUT : 000000 AMPM :0
 903 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0010 MINS_OUT : 000000 AMPM :0
1000 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 0 HRS_OUT : 0010 MINS_OUT : 000000 AMPM :0
1100 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0010 MINS_OUT : 000000 AMPM :0
1103 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0011 MINS_OUT : 000000 AMPM :0
1200 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 0 HRS_OUT : 0011 MINS_OUT : 000000 AMPM :0
1300 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0011 MINS_OUT : 000000 AMPM :0
1303 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0100 MINS_OUT : 000000 AMPM :0
1400 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 0 HRS_OUT : 0100 MINS_OUT : 000000 AMPM :0
1500 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0100 MINS_OUT : 000000 AMPM :0
1503 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0101 MINS_OUT : 000000 AMPM :0
1600 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 0 HRS_OUT : 0101 MINS_OUT : 000000 AMPM :0
1700 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0101 MINS_OUT : 000000 AMPM :0
1703 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0110 MINS_OUT : 000000 AMPM :0
1800 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 0 HRS_OUT : 0110 MINS_OUT : 000000 AMPM :0
1900 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0110 MINS_OUT : 000000 AMPM :0
1903 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0111 MINS_OUT : 000000 AMPM :0
2000 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 0 HRS_OUT : 0111 MINS_OUT : 000000 AMPM :0
```

The Verilog simulator is invoked for postlayout simulation using the following command:

```
ti_verilog timer_comp.v test_timer_comp.v -f
ba_timer.opts
```

Where:

timer_comp.v is the structural Verilog netlist

test_timer_comp.v is the modified test module which includes a prelayout or postlayout simulation switch and the call to the MDI Injector, as shown in Figure 4-35.

ba_timer.opts sets the prelayout/postlayout switch and tells the simulator to use maximum delays. These switches are shown in Figure 4-39.

Figure 4-39. Simulation Switches

```
+maxdelays
+define+prelayout
```

Using this options file, a prelayout simulation will be run; for a postlayout simulation, the second line should be commented out of the file with '//'.

Figure 4-40 shows the Verilog graphic output obtained from the postlayout simulation. The delay from CLK to HOURS_OUT is now 69 ns compared to a prelayout delay of 3 ns. This is clarified in Figure 4-41, which shows the **$monitor** output from the postlayout simulation.

Figure 4-40. Postlayout Simulation, CLK to HOURS-OUT Delay of 69 ns

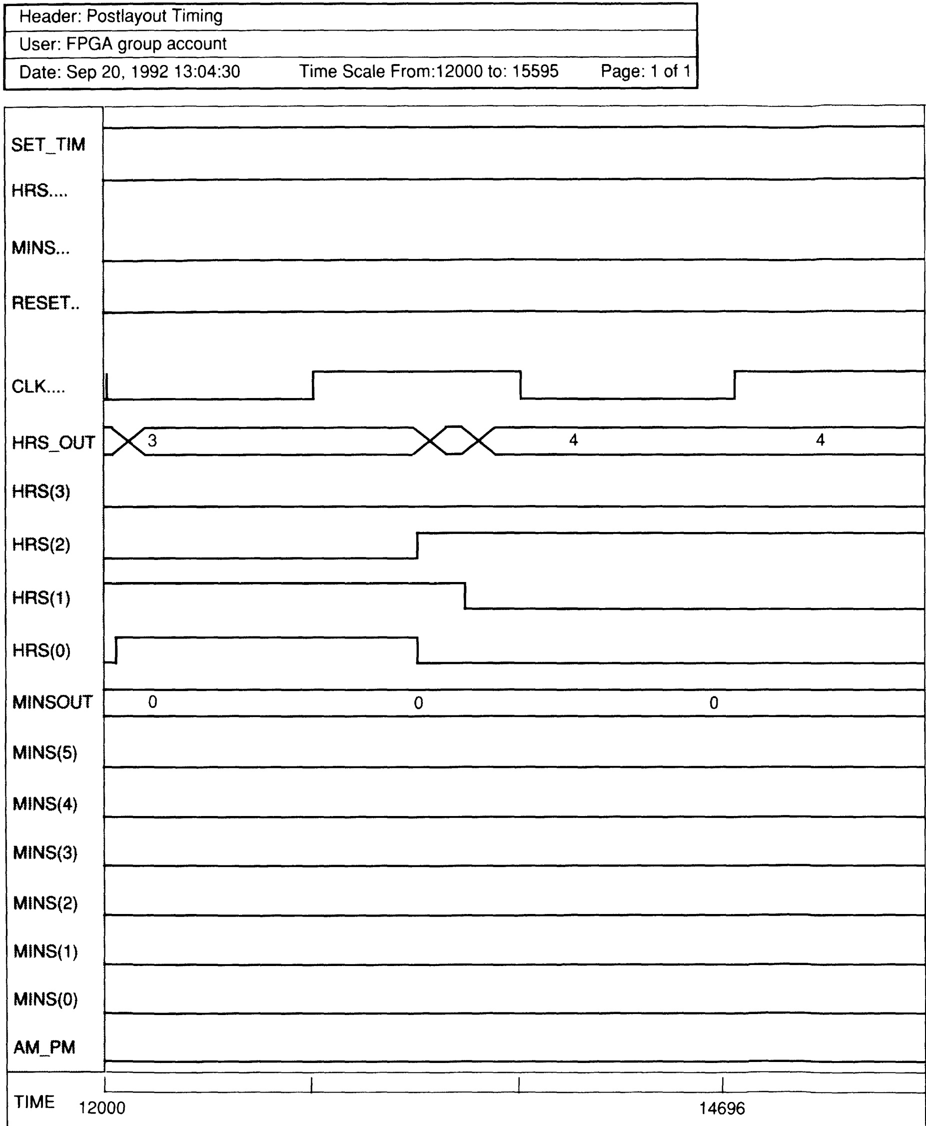


Figure 4-41. $Monitor Output for Postlayout Simulation

```
   0 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 0 CLK : 0 HRS_OUT : xxxx MINS_OUT : xxxxxx AMPM :x
 100 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : xxxx MINS_OUT : xxxxxx AMPM :x
 200 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 1 CLK : 0 HRS_OUT : xxxx MINS_OUT : xxxxxx AMPM :x
 300 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 1 CLK : 1 HRS_OUT : xxxx MINS_OUT : xxxxxx AMPM :x
 359 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 1 CLK : 1 HRS_OUT : xxxx MINS_OUT : xxxxx0 AMPM :x
 359 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 1 CLK : 1 HRS_OUT : xxxx MINS_OUT : 0xxxx0 AMPM :x
 359 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 1 CLK : 1 HRS_OUT : xxxx MINS_OUT : 0xxxx0 AMPM :0
 360 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 1 CLK : 1 HRS_OUT : xxxx MINS_OUT : 0xx0x0 AMPM :0
 360 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 1 CLK : 1 HRS_OUT : xxxx MINS_OUT : 0x00x0 AMPM :0
 361 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 1 CLK : 1 HRS_OUT : xxx0 MINS_OUT : 0000x0 AMPM :0
 361 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 1 CLK : 1 HRS_OUT : 0xx0 MINS_OUT : 0000x0 AMPM :0
 362 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 1 CLK : 1 HRS_OUT : 0xx0 MINS_OUT : 000000 AMPM :0
 363 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 1 CLK : 1 HRS_OUT : 00x0 MINS_OUT : 000000 AMPM :0
 369 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 1 CLK : 1 HRS_OUT : 0000 MINS_OUT : 000000 AMPM :0
 400 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 1 CLK : 0 HRS_OUT : 0000 MINS_OUT : 000000 AMPM :0
 500 SET_TIME : 0 HRS : 0 MINS : 0 RESET : 1 CLK : 1 HRS_OUT : 0000 MINS_OUT : 000000 AMPM :0
 600 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 0 HRS_OUT : 0000 MINS_OUT : 000000 AMPM :0
 700 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0000 MINS_OUT : 000000 AMPM :0
 760 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0001 MINS_OUT : 000000 AMPM :0
 800 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 0 HRS_OUT : 0001 MINS_OUT : 000000 AMPM :0
 900 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0001 MINS_OUT : 000000 AMPM :0
 961 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0000 MINS_OUT : 000000 AMPM :0
 968 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0010 MINS_OUT : 000000 AMPM :0
1000 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 0 HRS_OUT : 0010 MINS_OUT : 000000 AMPM :0
1100 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0010 MINS_OUT : 000000 AMPM :0
1160 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0011 MINS_OUT : 000000 AMPM :0
1200 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 0 HRS_OUT : 0011 MINS_OUT : 000000 AMPM :0
1300 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0011 MINS_OUT : 000000 AMPM :0
1361 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0010 MINS_OUT : 000000 AMPM :0
1361 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0110 MINS_OUT : 000000 AMPM :0
1369 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0100 MINS_OUT : 000000 AMPM :0
1400 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 0 HRS_OUT : 0100 MINS_OUT : 000000 AMPM :0
1500 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0100 MINS_OUT : 000000 AMPM :0
1560 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0101 MINS_OUT : 000000 AMPM :0
1600 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 0 HRS_OUT : 0101 MINS_OUT : 000000 AMPM :0
1700 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0101 MINS_OUT : 000000 AMPM :0
1761 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0100 MINS_OUT : 000000 AMPM :0
1768 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0110 MINS_OUT : 000000 AMPM :0
1800 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 0 HRS_OUT : 0110 MINS_OUT : 000000 AMPM :0
1900 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0110 MINS_OUT : 000000 AMPM :0
1960 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 1 HRS_OUT : 0111 MINS_OUT : 000000 AMPM :0
2000 SET_TIME : 1 HRS : 1 MINS : 0 RESET : 0 CLK : 0 HRS_OUT : 0111 MINS_OUT : 000000 AMPM :0
```

The TI-ALS contains an interactive static timing analysis tool used to analyze path delays. Setting the startset to CLOCK, the endset to OUTPAD and selecting LONGEST enables the user to determine the longest clock to output delay. The Timer output for these settings is shown in Figure 4-42. The longest path is CLK to HOURS_OUT1, which is 67 ns.

Figure 4-42. TI-ALS Timer Results

```
; TI Action Logic System, Release 2.1
; Welcome to the TI Action Logic System
; Timer, Release 1.11
;
; Initializing and loading design data...
; Loading delay data...
; Flattening design...
; Module statistics:
;        0        Sequential modules
;        160      Logical modules
;        16       IO modules
;        0        Clock modules
;        0        Sequential modules (after combining)
;        160      Logical modules (after combining)
;  1st longest path to all endpins
; Rank   Total   Start Pin        First Net      End Net         End Pin
;   0     66.9   IN_CLK:PAD       CLK            HRS_OUT1_1_     OUT_HRS1:PAD
;   1     60.6   IN_CLK:PAD       CLK            HRS_OUT1_2_     OUT_HRS2:PAD
;   2     59.2   IN_CLK:PAD       CLK            MNS_OUT1_1_     OUT_MINS1:PAD
;   3     59.2   IN_CLK:PAD       CLK            HRS_OUT1_3_     OUT_HRS3:PAD
;   4     58.4   IN_CLK:PAD       CLK            HRS_OUT1_0_     OUT_HRS0:PAD
;   5     58.4   IN_CLK:PAD       CLK            MNS_OUT1_4_     OUT_MINS4:PAD
;   6     57.9   IN_CLK:PAD       CLK            MNS_OUT1_3_     OUT_MINS3:PAD
;   7     57.4   IN_CLK:PAD       CLK            AM_PM1          OUT_AMPM:PAD
;   8     57.3   IN_CLK:PAD       CLK            MNS_OUT1_2_     OUT_MINS2:PAD
;   9     56.6   IN_CLK:PAD       CLK            MNS_OUT1_0_     OUT_MINS0:PAD
;  1st longest path to OUT_HRS1:PAD (falling) (Rank: 0)
; Total   Delay   Typ   Load   Macro    Start Pin        Net Name
;  66.9     9.9   Tpd     0    OUTBUF   OUT_HRS1:D       HRS_OUT1_1_
;  57.0    19.7   Tpd     6    DF1      TIMER_CORE/COUNT1/HRS_OUT_reg_1:CLK HRS_OUT_1_
;  37.3     9.6   Tpd     4    BUFA     TIMER_CORE/COUNT1/U92:A TIMER_CORE/COUNT1/n347
;  27.7     9.6   Tpd     2    BUFA     TIMER_CORE/COUNT1/U106:A TIMER_CORE/COUNT1/n334
;  18.1    18.1   Tpd     6    INBUF    IN_CLK:PAD       CLK
```

4.5.7 Summary

Using the high-level design flow described in this paper, it is possible to take an abstract textual description of a design, synthesize it to a gate-level description, and ultimately turn your design ideas into programmed silicon. This is a powerful tool for today's designer who is under increasing pressure to decrease time-to-market and to produce a right-the-first-time design.

4.6 ExpressV-HDL†

Hardware description languages (HDLs) are gaining increasing acceptance as the exploding complexity of designs and time-to-market pressures make it difficult for engineers to design at gate level. Automatic generation of VHDL from graphic input removes the burden of learning a new language for the hardware designer.

Generally, any HDL methodology requires engineers to write a text-based behavioral description which is then translated to gate level via synthesis tools. Such approaches offer higher productivity as a result of the greater levels of abstraction. However, for the hardware designer, a fundamental limitation of an HDL is that it is text-based.

First, this is not how an engineer intuitively thinks and designs. Second, text does not show the behavior of the system as clearly as an appropriate graphic.

With ExpressV-HDL, systems are described using 2 graphic languages: Activitycharts and Statecharts.

- Activitycharts

  The functional view of the model; these are a refinement of data-flow diagrams. They semantically and visually tie together the individual designs defined in the Statecharts. The Activitycharts illustrate the systems functional capabilities and show the data and control flows in the design.

- Statecharts

  The behavior implemented by the activities in the Activitycharts is defined in the Statechart View. Statecharts are a powerful extension of state transition diagrams, with the added notions of hierarchy and concurrency. The following are examples to illustrate these features.

  Figure 4-43 shows a simple state transition diagram and Figure 4-44 shows the equivalent Statechart.

† Contributed by Dung Tu, Technical Marketing, Texas Instruments Deutschland GmbH.

Figure 4-43. Simple State Transition

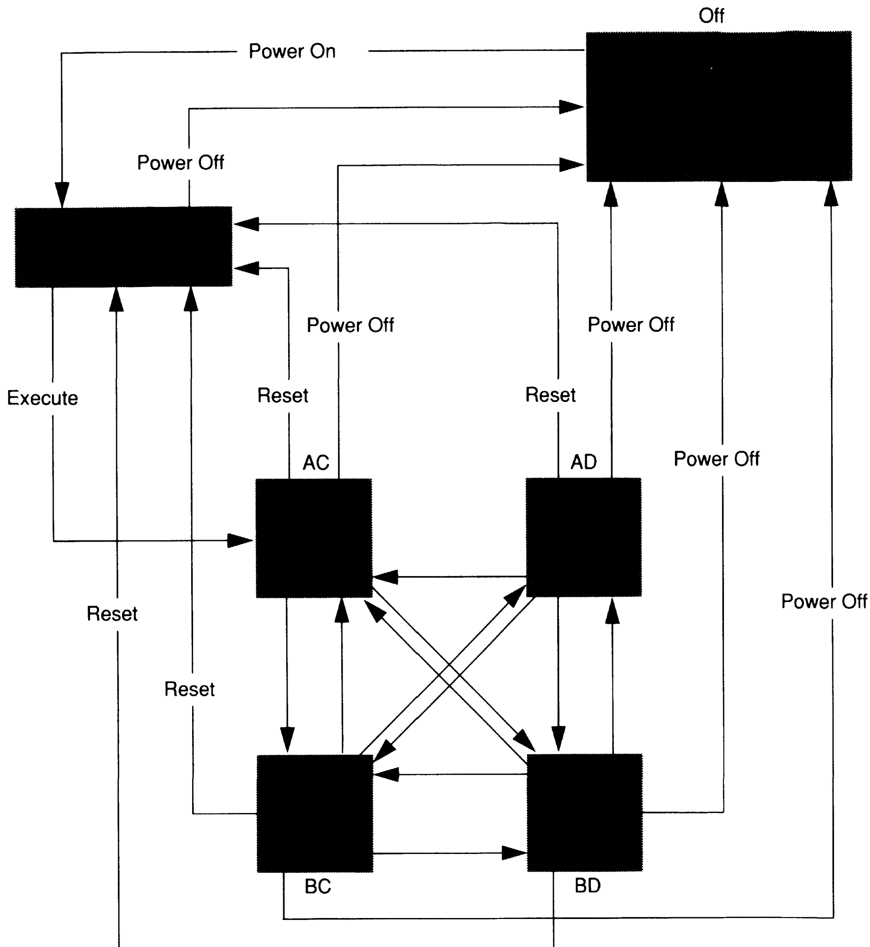
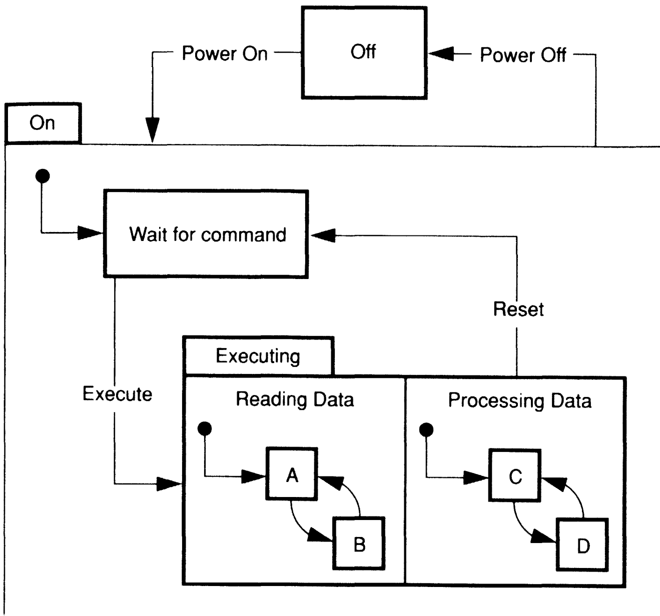


Figure 4-44. Equivalent Statechart



From the Statechart, there are clearly two states the system can be in. *Off* and *On*. There is now a single transition labeled *Power Off* from the *On* to *Off* States. Whenever this *power off* condition is produced, the system leaves the *On* state and enters the *Off* state, regardless of what is happening in the *On* state.

The obvious advantage is that the hierarchy has reduced the clutter by replacing the five transitions labeled *Power Off* with a single transition, this makes things easier to read. The hierarchy also has the effect of producing more efficient circuits after synthesis. Instead of having logic decode from all the different states when *power off* occurs, there is only one bit of decoding logic, thus producing a smaller and faster design.

The state *On* has two substates, *Wait for command* and *Executing*. The state *Executing* and its substates represent the four states, AC,AD,BC,BD, and their associated transitions. *Executing* is a concurrent state as indicated by the dotted line and consists of *Reading Data* and *Processing Data*. The broadcasting between the concurrent machines means that if *Processing Data* ever produces data or an event that *Reading Data* needs to operate on or relate to, then this is automatically sensed.

With the use of Statecharts, the perception of complexity is greatly reduced, thus making the design easier to read and understand.

Once Statecharts are created to describe the system behavior, the Statechart model may be validated using model execution. The model execution.tool, which can operate interactively or in batch mode, is executed before any code is generated.

Statecharts can also be created to model the system environment in order to ensure correct operation in the application. At code generation, these Statecharts can also be translated to HDL and used to test the final gate-level design produced after synthesis to ensure that it is correct.

In addition to pattern dependent checks, the analysis tools can also do exhaustive tests to check for race conditions or nondeterminants. Reachability and deadlock tests can also be performed.

Once the model has been verified, ExpressV-HDL automatically generates either behavioral or register transfer level (RTL) VHDL and Verilog code, without modification, directly from the Statecharts.

If behavioral code is produced, this can be used as a framework for further system development by enhancing and refining these behavioral models.If RTL output is produced, then this enables the translation to gate-level implementation through logic synthesis. The RTL code produced is optimized for the target synthesis tool to be used and formatted in accordance with the style guidelines of that particular synthesis tool in order to obtain the most efficient circuits after synthesis. Most major CAE vendor synthesis tools are supported.

By automatically generating vendor-specific RTL synthesizable HDL code, the code is implementation-independent and through logic synthesis can be targeted to either FPGA or ASIC implementation.

Also for large statechart models, during code generation it is possible to partition the design in order to allow a large design to be implemented across multiple FPGA devices. This partitioning can be based on the activitychart or any hierarchical branch of a statechart and does not require any chart modifications.

The underlying precision and semantic integrity of ExpressV-HDL assures that the generated code is compact, concise, and easily maintained. Hardware designers can now use an HDL design methodology and the power of synthesis tools without having to write and debug reams of handwritten code.

Chapter 5

PLD to FPGA Migration

This chapter discusses the TI logic integration tool (LIT); the proLogic compiler, a logic synthesis tool designed especially for FPGA designers familiar with Boolean equations and state machine entry; and Exemplar's Complete Optimization and Retargeting Environment (CORE1) software, providing FPGA-specific logic synthesis, optimization, and targeting.

5.1 TI Logic Integration Tool (LIT)†

The goal of logic integration is to reduce size, power, and cost by consolidating logic from lower to higher density devices. For example, designs consisting of TTL logic can be consolidated into single PLDs, FPGAs, or gate arrays.

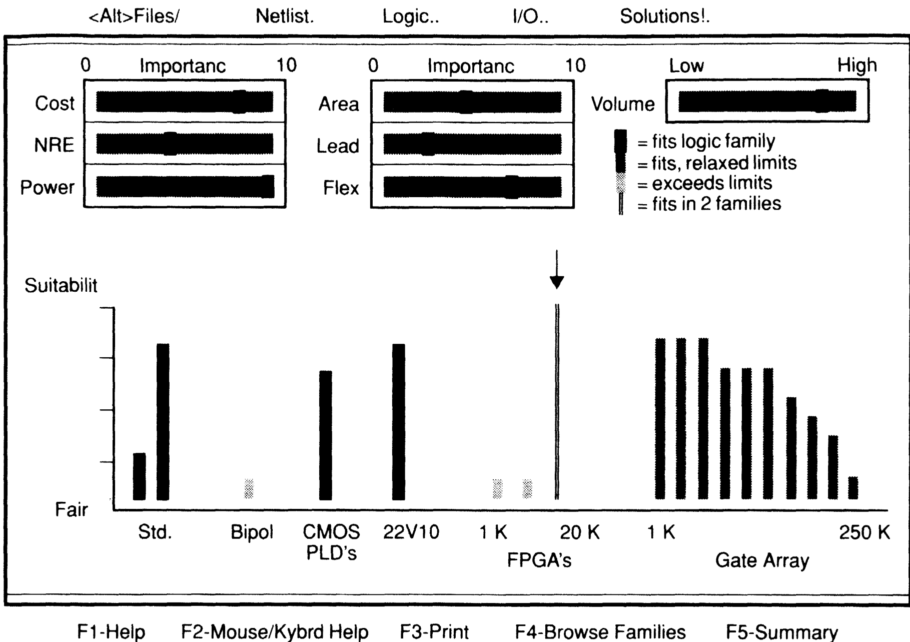
When you consider integration, ensure that the limitations of the new technology allow your design to function properly. Trade-offs in size, power, and cost must be both feasible and advantageous.

The LIT combines libraries containing a variety of data book information, such as I/O count, maximum input frequencies, drive capabilities, propagation delays, chip areas, and pricing for all logic families, enabling you to quickly execute a series of *what if* analyses. The resulting solution is an estimate of the most appropriate technology for the design.

To enter the design, choose standard devices from the libraries provided and answer questions concerning I/O and critical path timing. A small arrow on the display indicates the most appropriate technology for the design. Using graphic scroll bars, you can interactively adjust the importance of factors, such as cost, power, board area, NRE cost, lead time, design flexibility, and production volume (see Figure 5-1). Because the effects of moving the scroll bar factors are shown in real time, you can do dozens of *what if* calculations in minutes instead of days.

† Contributed by Travis Scheckel, FPGA Applications, Texas Instruments Incorporated.

Figure 5-1. Factors/Results Screen



You can also set maximum area and power constraints for your design. With the click of a function key, you can view a detailed report of the logic implementation.

The LIT includes a Multiple Logic Families feature that lets you split your design into two different technologies, which is useful when one technology is eliminated from consideration because of a single design criteria. Typically, if your design's critical path or output drive requirements are too high, the tool will either split the design into a faster technology or add buffer chips.

The LIT creates a model of your design based on real-world applications and typical data values of individual logic families; however, without a detailed design netlist, the model is only an estimate.

5.1.1 Hardware Requirements

The LIT runs on any IBM-compatible personal computer with 500 Kbytes of hard-disk space and RAM. The software supports CGA, EGA, VGA, and Hercules-compatible graphic adapters along with Microsoft-compatible mouse systems. You can choose either keyboard or mouse control.

5.1.2 Libraries

The libraries provided support a wide variety of logic families and design elements. The general purpose libraries include popular functions from the 74LS, 74AC, 74HC, 74S, 74F, and 74 series families. The PLD library includes the 16L8, 16R4, 16R6, 16R8, 20L8, 20R4, 20R6, 20R8, GAL16V8, GAL20V8, and the 22V10.

The libraries are ASCII text files that can be modified to add functions or to change data, such as pricing information for existing functions. Two dummy libraries let you collect frequently used parts.

Enter the following data to add a part:

- Part name
- Number of equivalent gates per function
- Cost per function
- Dip package area per function
- Surface mount area per function
- Power dissipation per function

For example, if a 74LS00 chip costs $0.12 and there are four NAND functions per chip, the cost per function will be $0.03.

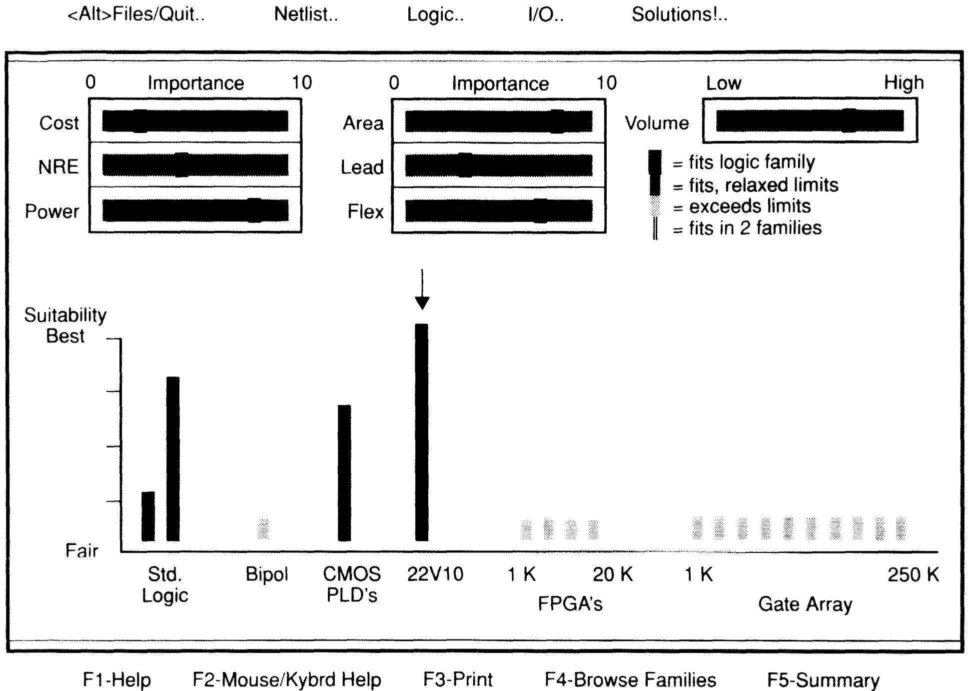
When you choose a function such as the 7404 hex inverter, you choose one inverter rather than all six inverters included on a single chip. If your design utilizes six inverters, you must enter the quantity after choosing the 7404 function.

5.1.3 Example

To help you visualize LIT capability, consider the critical path in an entered design consisting of three stages of sequential logic and two levels of intermediate combinational logic. The critical path allows a 20 ns delay between sequential stages and requires 40 inputs and 40 outputs with worst-case values of 82 MHz and 8 mA, respectively. The original logic is made up of random TTL functions and four PLDs with a total equivalent gate count of 2214.

After you set the scroll bar factors as desired, the LIT determines the appropriate technology for the design. As Figure 5-1 shows, the arrow, representing the solution, points to the 22V10.

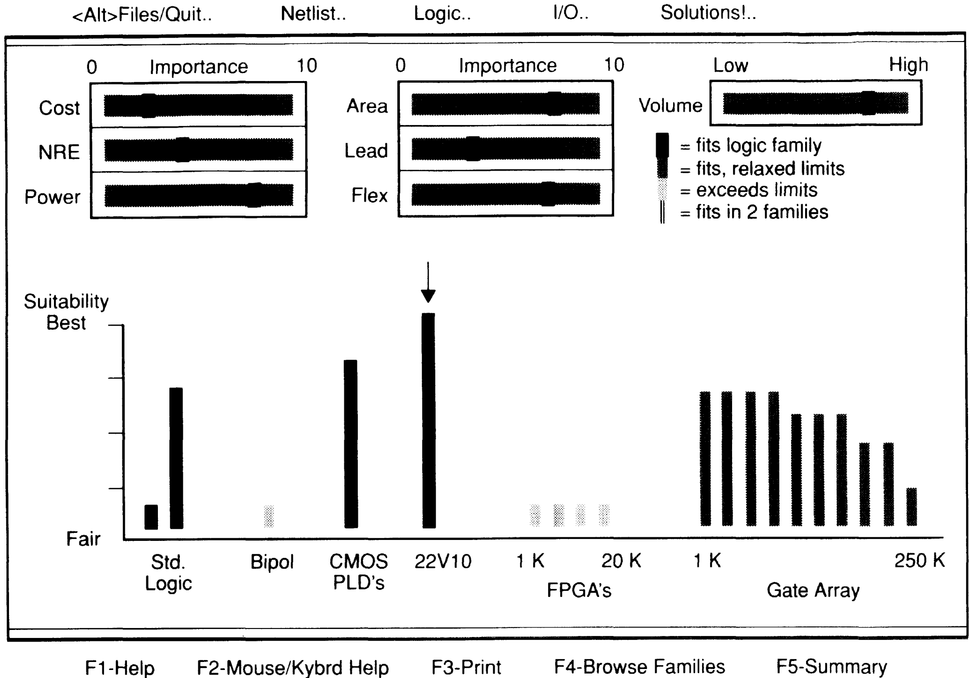
Figure 5-2. Results for a Single Logic Family with Maximum Constraints Enforced



Both FPGA and gate array technologies are eliminated because of the excessive input frequency of 82 MHz.

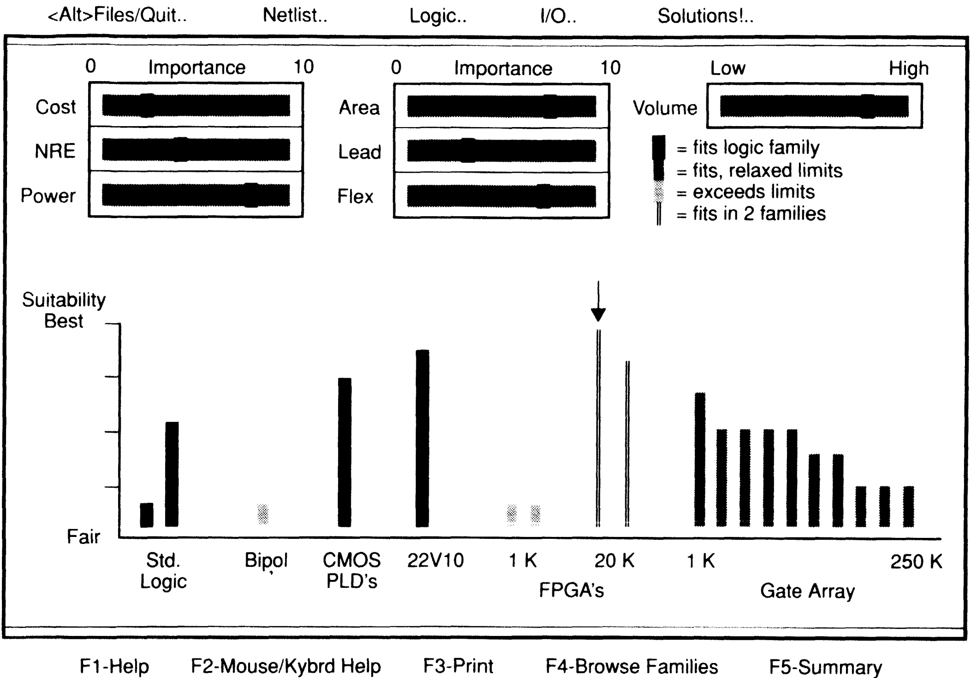
If the limits are relaxed 10 percent, gate arrays become a possible solution because they can handle input frequencies of up to 80 MHz (Figure 5-1). Even so, the 22V10 remains the most appropriate solution.

Figure 5-3. Results for a Single Logic Family with Maximum Constraints Relaxed



If the LIT splits the design into two logic families, however, the 4000-gate TPC1240 FPGA becomes the technology of choice to handle most of the design with a recommendation to implement PLDs for the fast inputs (Figure 5-1).

Figure 5-4. Results for Multiple Logic Families with Maximum Constraints Relaxed



5.2 ProLogic Compiler†

The proLogic compiler generates a JEDEC file along with a PALASM 2-compatible file in PDS format that serves as input to synthesis.

TI provides all the necessary software and hardware for a complete FPGA logic-synthesis design kit. For additional information, consult the documentation for each software package.

5.2.1 ProLogic Flow

As an example of how to create a prologic source file, verify functionality, and generate a `.pds` file, consider a simple 4-bit counter with synchronous clear; the counter is reset to 0 when CLR = 0 and counts up when CLR = 1.

First, create a subdirectory for the proLogic source and application files by entering the command:

```
md \designs\cnt41
```

Using an ASCII editor, create the proLogic source file in the new `cnt41` subdirectory by entering the command:

```
EDIT \designs\cnt41\cnt41.pld
```

The source file will be generated using the ASCII editor (Figure 5-5).

† Contributed by Dung Tu, Technical Marketing, Texas Instruments Deutschland GmbH.

Figure 5-5. The proLogic Source File

```
title {         Filename:       CNT4L.pld
                Function:       simple 4-bit binary counter
                Designer:       Dung Tu
                Date:           1.Sep.92
                Note:           22V10 is specified as device to produce a PDS file and
                                for simulation. The design is later converted to FPGA
                                using ALES.
}
include p22v10;     /* Specify a PLD device type                               */
include XOR.H;      /* To implement the "%" operator as Exclusive OR           */

/* Input pins */

define  CLK = pin1;
define  CLR = pin2;

/* Output pins */

define  Q0 = pin14;
define  Q1 = pin15;
define  Q2 = pin16;
define  Q3 = pin17;

/* Specify register outputs of the 22V10 as active high  */

Q0 = q;
Q1 = q;
Q2 = q;
Q3 = q;

/* The generic format for the n-th bit (Qn) of an n-bit counter is:          */
/* Q  = Q      %    (Q    &  Q    & ... & Q )
/*  n    n-1           n-1     n-2          0                                */
/* proLogic uses "Q.d" for a flipflop input and "Q.q" for the output */
/* Equations for a 4-bit counter:                                            */

Q0.d = !Q0.q & CLR;
Q1.d = (Q1.q %  Q0.q) & CLR;
Q2.d = (Q2.q % (Q1.q & Q0.q)) & CLR;
Q3.d = (Q3.q % (Q2.q & Q1.q & Q0.q)) & CLR;

/* Test vectors for proLogic functional simulator */
test_vectors {    CLK    CLR      Q3      Q2      Q1      Q0 ;  /* count */
                   C      0        L       L       L       L  ;  /* clear */
                   C      1        L       L       L       H  ;  /* 1 */
                   C      1        L       L       H       L  ;  /* 2 */
                   C      1        L       L       H       H  ;  /* 3 */
                   C      1        L       H       L       L  ;  /* 4 */
                   C      1        L       H       L       H  ;  /* 5 */
                   C      1        L       H       H       L  ;  /* 6 */
                   C      1        L       H       H       H  ;  /* 7 */
                   C      1        H       L       L       L  ;  /* 8 */
                   C      1        H       L       L       H  ;  /* 9 */
                   C      1        H       L       H       L  ;  /* 10 */
                   C      1        H       L       H       H  ;  /* 11 */
                   C      1        H       H       L       L  ;  /* 12 */
                   C      1        H       H       L       H  ;  /* 13 */
                   C      1        H       H       H       L  ;  /* 14 */
                   C      1        H       H       H       H  ;  /* 15 */
                   C      1        L       L       L       L  ;  /* 0 */
```

Referring to the following instructions, enter text in the source file.

- In the source file *title block*, enter the design name and comments.

Note:

The title block must be enclosed in braces ({ and }).

- In the *include block*, specify the PLD device type, from which you can generate a JEDEC file used downstream by the proLogic simulator to verify the functionality of the design using test vectors. The *include block* also contains the header files required for the compiler to function correctly. For example, the header file `XOR.H` is required to implement the operator % as an Exclusive OR.
- Both the *input pins block* and *output pins block* refer to the proLogic diagrams for each PLD device type. These blocks replace the pin number with a more meaningful signal name using the *define* statement. Due to compatibility with the PALASM 2 syntax, the pin definition is required only for simulation. Actual pin assignment for the FPGA is done downstream with the TI-ALS development software.
- In the *Specify register outputs block*, the register outputs of the 22V10 are configured as active-high or active-low. For simulation with proLogic the outputs must be specified as follows to ensure correct operation of the proLogic simulator:
  - Q0 = q specifies active-high.
  - !Q0 = q specifies active-low outputs.
- The *equation block* defines the design logic and can be a state machine or a truth table.
- The *test vectors block* is used by the proLogic simulator to verify the logic functions defined for the specified PLD. The test vectors define the inputs to the PLD and specify the expected output. The test patterns must be enclosed in braces ({ and }). The following naming convention applies:
  - 0 = drive input LOW
  - 1 = drive input HIGH
  - L = test output LOW
  - H = test output HIGH
  - C = drive input LOW-HIGH-LOW

Create the `.pds` file by entering the command:

```
lc cnt41 -i\prologic -p
```

The switch option `-i\prologic` tells proLogic to find the include and header files in the `\prologic` executable file directory.

To create the JEDEC file, enter the command:

lc cnt41 -i\prologic

If you create the batch file (`pl.bat`) using the commands:

lc %1 -i\prologic -p

lc %1 -i\prologic

you need only enter the command:

pl cnt41

ProLogic will generate the `cnt41.pds` file (Figure 5-6).

Figure 5-6. ProLogic-Generated `cnt41.pds` *File*

```
CHIP CNT4L PAL22V10
CLK CLR nc nc nc nc nc nc nc nc nc GND nc Q0 Q1 Q2 Q3 nc nc nc nc nc nc VCC
EQUATIONS
Q0 :=
                    /Q0 * CLR ;
Q1 :=
                    Q1 * /Q0 * CLR
                  + /Q1 * Q0 * CLR ;
Q2 :=
                    /Q2 * Q1 * Q0 * CLR
                  + Q2 * /Q0 * CLR
                  + Q2 * /Q1 * CLR ;
Q3 :=
                    /Q3 * Q2 * Q1 * Q0 * CLR
                  + Q3 * /Q0 * CLR
                  + Q3 * /Q1 * CLR
                  + Q3 * /Q2 * CLR ;
Q3.TRST =
                    VCC ;
Q2.TRST =
                    VCC ;
Q1.TRST =
                    VCC ;
Q0.TRST =
                    VCC ;
```

To use the proLogic simulator for design verification, enter the command:

ls cnt41 -a\prologic\p22v10.lxa

The `-a` option tells proLogic to use the architecture description file `p22v10.lxa` located in the `\prologic` directory. The simulator uses this file along with the JEDEC file to compile an optimized device model prior to the actual simulation.

The test results are written into the `cnt41.tst` test result file (Figure 5-7).

Figure 5-7. Example Test Result File (`cnt41.tst`)

```
proLogic Simulator
Texas Instruments V2.0
Copyright (C) 1991 Prologic Systems

Architecture Description: \prologic\p22v10.lxa
JEDEC Fuse Information:   CNT4L.jed
JEDEC Test Vectors:     CNT4L.jed

V01  C0NN NNNN NNNN NLLL LNNN NNNN
V02  C1NN NNNN NNNN NHLL LNNN NNNN
V03  C1NN NNNN NNNN NLHL LNNN NNNN
V04  C1NN NNNN NNNN NHHL LNNN NNNN
V05  C1NN NNNN NNNN NLLH LNNN NNNN
V06  C1NN NNNN NNNN NHLH LNNN NNNN
V07  C1NN NNNN NNNN NLHH LNNN NNNN
V08  C1NN NNNN NNNN NHHH LNNN NNNN
V09  C1NN NNNN NNNN NLLL HNNN NNNN
V10  C1NN NNNN NNNN NHLL HNNN NNNN
V11  C1NN NNNN NNNN NLHL HNNN NNNN
V12  C1NN NNNN NNNN NHHL HNNN NNNN
V13  C1NN NNNN NNNN NLLH HNNN NNNN
V14  C1NN NNNN NNNN NHLH HNNN NNNN
V15  C1NN NNNN NNNN NLHH HNNN NNNN
V16  C1NN NNNN NNNN NHHH HNNN NNNN
V17  C1NN NNNN NNNN NLLL LNNN NNNN

No errors detected with 17 Test Vectors.
```

The test result file indicates that no errors are detected, indicating that the logic equations work correctly.

5.2.2 Viewlogic Flow

This section shows you how to create block symbols and a top-level schematic and generate a netlist with input and output buffers for the complete design.

The example equations shown in Section 5.2.1 describe the function of the 4-bit counter but exclude the FPGA input and output buffers. To complete the design, use a CAD tool to create a symbol with the same name as your equation file (e.g., `cnt41`), then create a top-level schematic (`cnt41t`) that includes the symbol with the required input and output buffers.

All Viewlogic projects are presumed to be saved under the `\proj` directory:

- `\proj\cnt41t` (EDIF netlists, simulation netlists, etc.)
- `\proj\cnt41t\sch` (schematic files)
- `\proj\cnt41t\sym` (symbol files)
- `\proj\cnt41t\wir` (wirelist files)

Use the Viewfile utility for every individual design to create the required subdirectories so that your CNT4LT design will be stored under `\proj\cnt41t`.

Prologic and TI-ALES projects are stored under the directory `\designs`:

- `\designs\cnt41` (proLogic files)
- `\designs\cnt41t` (TI-ALS files generated by MAKEADL or TI-ALS)

Your `cnt41` proLogic files are saved under the subdirectory:

```
\designs\cnt41
```

For the top-level design CNT4LT, all TI-ALS files will be saved in the following subdirectory automatically created by TI-ALS:

```
\designs\cnt41t
```

To create a directory structure for Viewlogic that works well with proLogic and ALES, invoke Viewlogic by entering the command:

wv

Use Viewfile to create the project:

```
Window | Open | Viewfile
Project | Create                  Project name \proj\cnt41t
Set | Project                     cnt41t
```

Convert the EDIF netlist to a Viewlogic wirelist using EDIFNETI:

1) Copy the EDIF netlist `cnt41.edn` from the `\designs\cnt41` subdirectory to `\proj\cnt41t`.

2) Create a wirelist by entering the command:

   edifneti cnt41.edn

The EDIF netlist reader converts the EDIF netlist to a wirelist and saves the netlist under the subdirectory:

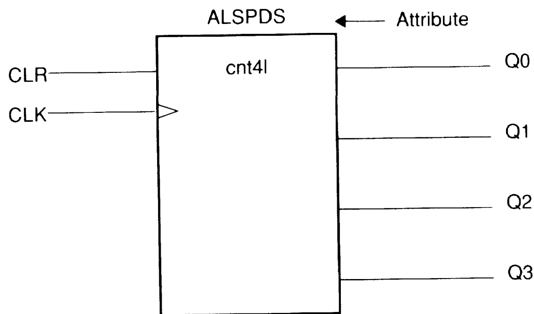
```
\proj\cnt41t\wir
```

Create a symbol in Viewdraw as follows:

```
Window | Open | Viewdraw | Symbol      Symbol name: cnt41
Change | Block | Sheet | z-wxh         Block width: 200
                                       Block height: 200
Add | Box                              Draw a box
Add | Pin                              Draw the pins
Add | Label                            Name the pins
```

The symbol, shown in Figure 5-8, has two input pins (CLR, CLK) and four output pins (Q0, Q1, Q2, Q3).

Figure 5-8. Symbol Created in Viewdraw



NOTE: cnit4l, entered by using `ADD | Text`, is a comment and not a label.

Add the *ALSPDS* attribute to the symbol you create using Viewdraw to inform TI-ALS that you are working with equations rather than a schematic:

```
View | Out        to get more space
Add  | Attr       Enter: ALSPDS
```

The symbol and wirelist names must be identical for the symbol and wirelist to be linked together.

Create a top-level schematic and generate a top-level wirelist with Viewdraw as follows:

```
Window | Open | Viewdraw | Schematic  Name: cnt41t
Add | Comp      Name: cnt41
Add | Comp      Add the input/output buffers
Add | Net       Connect the components
etc...
File|Write      Create top level wirelist
```

Use the term *INBUF* as input buffer for CLR, *CLKBUF* for CLK, and *OUTBUF* for the outputs.

You have now generated a complete wirelist that includes your 4-bit counter plus the FPGA input/output buffers and are finished with the Viewlogic flow.

5.2.3 TI-ALS Flow

The following steps show you how to use TI-ALS to generate an ADL netlist, specify the FPGA device, run place and route, and generate a fuse map to program your FPGA design.

1) To generate an ADL netlist, convert the wirelist to ADL format by entering the commands:

   ```
   cd \proj\cnt4lt
   vl2adl cnt4lt
   ```

   MAKEADL automatically creates the subdirectory `\designs\cnt4lt`, where all the required files for TI-ALS are saved.

2) To specify an FPGA device, invoke the TI-ALS main menu by entering the command:

   ```
   als cnt4lt
   ```

   Select the menu items:

   `Project | Device | TPC10 | TPC1010A | 68PLCC`

Note:

This step can only be done within the TI-ALS main menu.

3) To run Validate and place and route, first select the following command from the TI-ALS main menu:

   `Validate | Run`

   The `Validate` command checks for FPGA design rule violations and shows the statistics for the modules used. To run place and route, select the command:

   `Config | Run`

   Automatic pin assignment is used so that Pinedit is not required.

4) To perform a timing analysis using the TI-ALS timer, select the following command from the TI-ALS main menu:

   `Timer | Run`

Specify temperature and voltage by clicking on the menu items. For the longest delay time, select the command:

```
Timing | Longest
```

The *Longest Path Selection* dialog box will be displayed.

If you click OK to use the default pin sets, the timer will show you the longest delay paths, as shown in Figure 5-9.

Figure 5-9. TI-ALS Timer Longest Delay Paths

```
; 1st longest path to all endpins
; Rank Total  Start pin    First Net   End Net     End pin
; 0     36.4  U1/I8:CLK    IQ1         U1/N11      U1/I0:S
; 1     36.3  U1/I10:CLK   IQ0         U1/N9       U1/I9:S
; 2     29.9  U1/I10:CLK   IQ0         U1/N7       U1/I8:S
; 3     26.1  U1/I10:CLK   IQ0         U1/N6       U1/I8:A
; 4     17.4  U1/I10:CLK   IQ0         IQ0         U1/I10:S
```

The signal names displayed on your screen may be different, depending on the labels you have selected for your signals.

To obtain all the delay elements summed up to the total delay of the longest path (Rank 0), select from the TI-ALS Timer menu the command:

```
Timer | Expand
```

In the *Expand Path Selection* submenu, enter Rank 0. The timer will show you the delay elements (Figure 5-10).

Figure 5-10. TI-ALS Timer Delay Elements

```
;  1st longest path to U1/I0:S (rising) (Rank: 0)
; TotalDelayTyp Load  Macro   Start pin    Net name
;  36.36.4 Tsu   0    DFM     U1/I0:S
;  29.98.0 Tpd   1    AX1     U1/I1:A      U1/N11
;  21.98.3 Tpd   2    NAND2   U1/I12:A     U1/N10
;  13.615.4 Tcq  7    DFM     U1/I8:CLK    IQ1
;  -1.8-1.8 Psk  8            U1/I0:CLK    ICLK
```

5) To create the fuse map required to program an FPGA, select from the TI-ALS main menu the command:

```
Fuse | Run
```

A fuse file named `cnt4lt.fus` will be generated that you can use with an Activator 1 or Activator 2 programming unit to program the FPGA device you have selected.

To invoke a complete TI-ALS run, exit the TI-ALS main menu and enter from DOS the command :

```
alsrun cnt4lt
```

Your design is now complete.

5.2.4 A Complex Circuit With More Pins Than the Supported PLDs

Complex FPGA design flows are similar to simple design flows but may need to be broken down into several function blocks, each of which can be fitted into a PLD; e.g., a 22V10.

If a design will not fit into a 22V10, (for example, a 13-bit loadable counter with 13 load inputs), you can build a 13-bit counter by cascading an 8-bit counter with a 5-bit counter. Use proLogic to design your own 8-bit cascadable counter, or use the TA269 8-bit counter from the FPGA schematic library and design your own 5-bit cascadable counter.

When you use your CAD tool to connect the 8-bit counter with the 5-bit counter, the CAD tool will create a netlist for the 13-bit counter. With MAKEADL, you can then create the ADL netlist for the 13-bit counter.

Figure 5-11 shows the source file for a 5-bit loadable and cascadable counter using proLogic.

If the proLogic simulator shows no errors for each function block and the timing is not critical, your design will normally work. Perform a timing analysis using the TI timer.

If your design is complex and the timing is critical, you may want to do a back-annotated simulation. In this case, you will require a simulator that allows you to do a postlayout timing simulation for the complete design. Please contact TI for assistance.

Figure 5-11. Source File for 5-Bit Loadable and Cascadable Counter Using ProLogic

```
title {             Filename:     cnt51.pld
                    Function:     5-b it loadable counter with synchronous
               clear, active high carry -in and carry-out
                    Designer:     Dung Tu
                    Date:         31.Aug.92
                    Note:         22V10 is specified as device to produce a
               PDS file and for simulation. The design is later converted to
               FPGA using ALES.
}
include p22v10;
include XOR.H;
define CLK = pin1;
define CLR = pin2;
define   LD = pin3;    /* load control pin, load when LD=H */
define   CI = pin4;    /* carry-in for cascading */
/* load inputs */
define  P0 = pin5;
define  P1 = pin6;
define  P2 = pin7;
define  P3 = pin8;
define  P4 = pin9;
/* counter outputs */
define  Q0 = pin14;
define  Q1 = pin15;
define  Q2 = pin16;
define  Q3 = pin17;
define  Q4 = pin18;
define  CO = pin22;    /* carry-out for cascading */
/* 22V10 outputs defined as active-high */
Q0 = q;
Q1 = q;
Q2 = q;
Q3 = q;
Q4 = q;
/* 5-bit counter equations */
Q0.d = ( LD & P0
       | !LD & (Q0.q % CI)) & CLR;
Q1.d = ( LD & P1
       | !LD & (Q1.q % (Q0.q & CI))) & CLR;
Q2.d = ( LD & P2
       | !LD & (Q2.q % (Q1.q & Q0.q & CI))) & CLR;
Q3.d = ( LD & P3
       | !LD & (Q3.q % (Q2.q & Q1.q & Q0.q & CI))) & CLR;
Q4.d = ( LD & P4
       | !LD & (Q4.q % (Q3.q & Q2.q & Q1.q & Q0.q & CI))) & CLR;
CO   =  !LD & Q4.q & Q3.q & Q2.q & Q1.q & Q0.q & CI;
```

```
test_vectors {
CLK CLR LD  CI  P4  P3  P2  P1  P0  Q4  Q3  Q2  Q1  Q0  CO;
C   0   X   X   X   X   X   X   X   L   L   L   L   L   L; /* clear counter  */
C   1   1   X   1   0   1   0   1   H   L   H   L   H   L; /* load 10101     */
C   0   1   X   1   0   1   0   1   L   L   L   L   L   L; /* clear          */
C   1   0   1   X   X   X   X   X   L   L   L   L   H   L; /* count 1        */
C   1   0   1   X   X   X   X   X   L   L   L   H   L   L; /* count 2        */
repeat 11 {
C   1   0   1   X   X   X   X   X   X   X   X   X   X   L; /* repeat 11 times */
                                                           /* outputs not tested */
}
C   1   0   1   X   X   X   X   X   L   H   H   H   L   L; /* now count 14   */
C   1   0   1   X   X   X   X   X   L   H   H   H   H   L; /* count 15       */
C   1   0   1   X   X   X   X   X   H   L   L   L   L   L; /* count 16       */
C   1   0   1   X   X   X   X   X   H   L   L   L   H   L; /* count 17       */
repeat 13 {
C   1   0   1   X   X   X   X   X   X   X   X   X   X   L; /* repeat 13 times */
                                                           /* outputs not tested */
}
C   1   0   1   X   X   X   X   X   H   H   H   H   H   H; /* now count 31   */
                            /* carry-out is high because Q0 .. Q4 are high */
C   1   0   1   X   X   X   X   X   L   L   L   L   L   L; /* count 0        */
}
```

5.3 FPGA Logic Synthesis Using Exemplar[†]

5.3.1 Introduction

Exemplar's Complete Optimization and Retargeting Environment (CORE1) software provides logic synthesis, optimization, and targeting of FPGA designs captured in various netlist formats to FPGA netlist format with little or no hand-editing; thus, you can move to a faster, larger, and more cost-effective solution without having to recapture your design on a different CAD tool.

Exemplar Core1 provides read and write of EDIF 2.0.0 netlist and is compatible with other synthesis products in widespread use.

After a design is successfully translated to FPGA EDIF or ADL format, you can use it as a netlist input to the TI ASIC design flow, which offers more cost-effective solutions for volume production.

5.3.2 Software Description

This section describes the user-interface and the features that are of most use in the translation path from Xilinx to TI-FPGA.

CORE1 version 1.20 can be used from the graphical user interface (GUI) or the command line. To invoke the GUI, enter the command:

```
exemplar &
```

To invoke from the command line, enter the command:

```
fpga design_in.ext design_out.ext <switches>
```

Where

design_in and *design_out* represent the relevant technology.

The GUI run-time display provides initial module and gate counts, longest path delay information, design statistics after buffering for excessive fanout, and the netlist as each optimization pass is completed. The area and speed trade-off made by each optimization pass is shown graphically so that you can select the desired result instead of the software.

The software utilizes optimization algorithms designed for large, entirely combinational circuits containing many logic levels. Existing tools have traditionally had a problem with these kinds of circuits, particularly if the circuit is substantially arithmetic. Excessive memory is required to handle

[†] Contributed by Chris Ward, FPGA CAD Support, Texas Instruments Ltd.

the large number of product terms resulting from the collapsing logic levels necessary before optimization can begin.

Exemplar software partitions these designs before optimizing to reduce the number of product terms and save memory and CPU time. Other areas of improvement include the automatic translation of internal 3-stateable gates to combinational equivalents (where appropriate), I/O cell insertion, and a quick-optimization option for evaluating designs.

5.3.3 Controlling Optimization

The following switches provide set optimization parameters:

- *area/delay*
  - The *area* switch optimizes for the lowest gate/module count by saving the output file netlist with the lowest module count. Minimum effort is made to reduce average delays.
  - The *delay* switch allows the output file netlist with the shortest worst-case delay path to be saved.
- *quick/normal/remap*
  - The *quick* switch allows a single-pass optimization taking only a few minutes to complete, even on designs as large as 2000 modules, which is useful for evaluations of circuit optimization and for early decisions on target device.
  - The *normal* switch takes longer to run but may invoke as many as 10 optimization attempts, each with a different algorithm. The best optimization result is written to the output file specified by you.
  - The *remap* switch prevents design optimization, leaving instance names unchanged; however, buffers on nets with excessive fanout will be inserted, as will I/O buffers, if the *chip* switch is set.
- *chip/macro*
  - The *chip* switch causes the software to provide input buffers on nets with no source and output buffers on nets with no load. Existing I/O buffers are maintained.
  - The *macro* switch removes any I/O buffers found.
- *small/large*
  - The *large* switch partitions the design before optimization, allowing large combinational and arithmetic designs to be optimized without using prohibitive amounts of memory. Nonpartitioning synthesis software can consume up to 100 Mbytes of memory optimizing a 1500-module arithmetic design.

  - The *small* switch does not partition the design before optimization.

5.3.4 Other Features

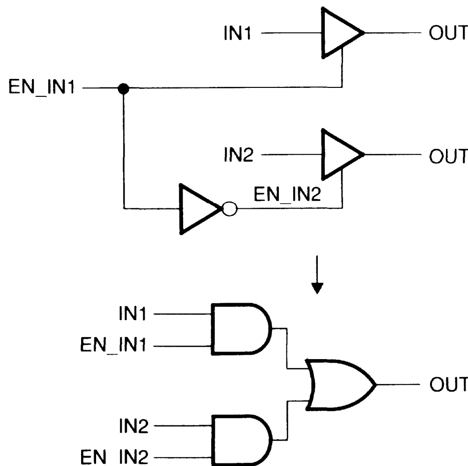
The *additional options* screen provides control of register transformation (`-transform`) and 3-state mapping (`-tristate`). The `-transform` option provides mapping of latches and flip-flops when the target library has no exact functional equivalents. Differences must be limited to preset or clear inputs; the component will be connected to use whichever is available.

The `-tristate` mapping option is essential for targeting FPGA, because no internal 3-stateable cells are available in this technology. When running using this option, include the command line argument:

`-tristate_map=nopreserve_z`

This will result in the mapping shown in Figure 5-12.

Figure 5-12. Mapping Internal Three-States to Combinational Equivalents



The overhead incurred when mapping from Xilinx to FPGA is of the order of one logic module per 3-state.

5.3.5 Migration to ASIC

Although Exemplar is an effective migration solution for TI ASIC customers, no test compiler features are supported. For example, to migrate from FPGA to ASIC, proper fault coverage must be obtained for the design, which means the insertion of scan chains and ATPG, particularly for short development timescales. For information on ASIC migration, see Chapter 6.

5.3.6 Design Examples

This section contains two examples showing the effect of circuit architecture on synthesis and optimization. Both examples are large Xilinx.`xnf`-to-FPGA.`adl` format translations.

Statistics for module usage, combinational and sequential ratio, and I/O counts are recorded in log files shown in Figure 5-13 through Figure 5-15.

Figure 5-13. Summary of Remapping the Targeting 10 Series

```
FPGA compilerMon Apr 27 09:33:15 1992
          Estimated              Output
Pass Area      Delay        CPU    File
(modules)       (ns)    min:sec
Initial        1304      100.2      00:23    cct_1.edf
1              1268       95.2      08:25    cct_2.edf
       Resource Use Estimate
Design: pga
Technology: act1
File: pga.adl
Area: 1268.0
Critical Path: 95.2  ns
Device             Area            Registers           i/o
   avl/usd /pct       avl/usd/pct        avl/usd/pct
1010            295/1268/430%      147/342/233%       57/ 66/116%
1020            546/1268/232%      273/342/125%       69/ 66/ 96%
            Delay Summary
Node:                   Slack      Arrival        Required     Load
               rise    fall    rise    fall
NOD1_1-000026       :  21.40   73.80   73.80   95.20   95.20    1.00
NOD1_1-000014:24.6070.6070.6095.2095.201.00
                             :
EXT_SIGSYNPAD:  87.00    8.20    8.20   95.20   95.20    1.00
OPT1_1-000007       :  87.00    8.20    8.20   95.20   95.20    1.00

        Cell Usage Summary
Cell             Uses           Cost              Total
AND2          57 uses(s)    1.00 modules    57.00 modules
AND2A         26 uses(s)    1.00 modules    26.00 modules
                 :
OR4A          16 uses(s)    1.00 modules    16.00 modules
TA157          2 uses(s)    1.00 modules     2.00 modules
                                            -------
                                Total = 1268.00
```

Figure 5-14. Summary of Optimizing the Targeting 12 Series

```
Exemplar Logic Synthesis System Mon Sep 14 17:42:38 1992
                    Estimated            Output
Pass             Area      Delay        CPU   File
   (modules)         (ns)    min:sec
1                1103      140.8      07:17   ccta_app_1.adl
2                1085      143.2      07:00   ccta_app_2.adl
             Resource Use Estimate
Design: cct_app_2
Technology: act2
File: cct.xnf
Area: 1085.0
Critical Path: 143.2 ns
Device                 Area             Registers           i/o
         avl/usd /pct       avl/usd/pct        avl/usd/pct
A1225            451/1085/241%      341/  0/  0%       82/ 68/ 83%
A1240            684/1085/159%      565/  0/  0%      104/ 68/ 65%
A1280           1232/1085/ 88%      998/  0/  0%      140/ 68/ 49%
                   Delay Summary
Node:                     Slack      Arrival        Required     Load
                         rise    fall    rise    fall
DP/1_1-000009       :  43.90   99.30   99.30 143.20 143.20    1.60
DP/1_1-000001       :  46.30   96.90   96.90 143.20 143.20    1.60
DP/1_1-000025       :  46.30   96.90   96.90 143.20 143.20    1.60
                                :
                                :
AP/1_1-000009       : 135.10    8.10    8.10 143.20 143.20    1.60
AP/1_1-000001       : 135.10    8.10    8.10 143.20 143.20    1.60
              Cell Usage Summary
Cell                Uses           Cost              Total
AND2            22 uses(s)    1.00 modules    22.00 modules
AND2A           18 uses(s)    1.00 modules    18.00 modules
AND3             6 uses(s)    1.00 modules     6.00 modules
                                :
                                :
XA1              1 uses(s)    1.00 modules     1.00 modules
XOR              4 uses(s)    1.00 modules     4.00 modules
                                                -------
                                     Total = 1085.00
```

Figure 5-15. Summary File Targeting 10 Series

```
FPGA compiler                    Tue May 19 15:09:56 1992

           Estimated             Output
Pass            Area       Delay         CPU    File
(modules)        (ns)     min:sec
1               1937        83.8       00:07    bar_opt_1.adl
2                805        98.4       04:41    bar_opt_2.adl

  Resource Use Estimate
            Design: bar_opt_3
        Technology: act1
              File: bar.adl
              Area: 805.0
Critical Path: 98.4 ns

Device               Area             Registers             i/o
      avl/usd /pct        avl/usd/pct        avl/usd/pct
1010           295/ 805/273%      147/ 69/ 47%        57/ 69/121%
1020           546/ 805/147%      273/ 69/ 25%        69/ 69/100%

           Delay Summary
  Node:          Slack       Arrival        Required     Load
              rise    fall    rise    fall
NOD450004  :   0.00   98.40   98.40   98.40   98.40     1.00
NOD970000  :   0.00   98.40   98.40   98.40   98.40     1.00
                                   .
                                   .
                                   .
Cell              Uses          Cost              Total
AND2           16 uses(s)    1.00 modules    16.00 modules
AND2A           3 uses(s)    1.00 modules     3.00 modules
                   .
                   .
                   .
```

5.3.6.1 Interface Controller

The first design example is an interface controller for a telecom application with an average mix of combinational and sequential logic and a large number of internal 3-stateable devices.

Initial attempts at synthesis/optimization cause the tool to report problems complaining of 3-states that are not translatable.

The command is entered:

`fpga pga.xnf pga.adl -so=xi3 -ta=act1 -tri -rem`

where `-tri[state_map]` provides combinational logic to be inserted instead of any 3-state logic and `-rem[ap]` provides no optimization.

The release notes supplied with the installation document two options available on the command line for 3-state mapping:

- ❏ `preserve_z`
- ❏ `nopreserve_z`

Using the `nopreserve_z` option, a combinational circuit is constructed to replace each 3-state as follows:

```
fpga      pga.xnf      pga.adl      -so=xi3      -ta=act1
-tristate_map=nopreserve_z
```

This circuit will remove the 3-states from the `.xnf` and substitute combinational circuitry instead.

The smallest gate count produced is1268, which is excessive for a TPC10 series FPGA with a propagation delay of 95.2ns. Even after optimization for delay is attempted, the module count increases to 1322 and the delay decreases to 88.2 ns. No significant impact is made by optimizing `.adl` to `.adl`. The lowest count is1235 modules, but in most cases is up to1329.

Providing .adl from Exemplar is not the most useful entry point to TI-ALS. The file does not have any headers, and the `.ipf` and `.crt` files are not available and must be created by hand. The files should contain only the following four lines:

```
;header
;endheader
def <design_name>.
end.
```

After creating the files and before invoking TI-ALS, run the TI-ALS program by entering the command:

convert *design_name*

The most convenient output to generate from Exemplar is `.edif`. Using the `-quick` optimization, the `.adl` for this design is translated and written out as `.edif`. Some redundant logic is automatically removed in the process, decreasing the module count to1258.

The netlists may need to be hand-edited to handle cell mismatches across technologies. For example, an equivalent for the *OSC* oscillator cell in Xilinx is not available in the TPC10 series or TPC12 series library; however, mapping to a simple user-defined soft macro will solve this problem and other similar ones.

5.3.6.2 Barrel Shifter

The second design example is a barrel shifter that is entirely combinational, containing approximately 1900 modules and12k equivalent gates. The Input file is `bar.adl`.

Large combinational circuit designs traditionally have not been able to be synthesized because of an arithmetic structure unsuited to early algorithms, such as MIS and Espresso. As shown in Figure 5-15, Exemplar Core1 successfully synthesizes and optimizes large designs with nonsequential logic.

Optimization using Exemplar on a Sun Sparc 2 takes approximately one hour, compared to over 75 Mbytes and several hours required to optimize an HP/Apollo system without Exemplar.

After optimizing, the resulting `.adl` is optimized again using `-quick`, which results in 537 modules.

Using Examplar on a design that is the purely combinational with repeated functional blocks reduces well within an acceptable time and utilizes the partitioning algorithm efficiently.

Chapter 6

FPGA to ASIC Migration

This chapter describes the automated methodology available for migrating field programmable logic (FPL) to ASIC gate array or standard cell technology.

In addition, FPGA to ASIC migration options are discussed along with design rules for testability.

6.1 Migrating Programmable Logic to ASIC †

The FPL netlist is the input to the migration flow. The translation software replaces existing flip-flops with equivalent scan flip-flops and connects a scan chain. ATPG software then produces high-fault-coverage test patterns.

Timing reports and workstation schematics of the ASIC are available after migration for timing verification and device simulation before manufacturing. The methodology reduces execution time for FPL migration from man-months to man-days.

6.1.1 Motivation

The automated FPL migration path to gate array or standard cell ASIC technology reduces design cycle time and long-term production costs. Migration methodology supports TI FPGA migration; additional interfaces support Xilinx FPGA and Altera EPLD migration. Among the advantages programmable logic offers for digital designs in development are:

- Highly integrated, high-performance functions
- Rapid functional verification and modification
- Inexpensive PC-based development tools
- Low NRE costs per device

Cost concerns take precedence after a design moves into production. The cost per gate of FPL devices is typically an order of magnitude greater than for an ASIC. Yet NRE costs for ASIC manufacturing are typically two orders of magnitude greater than the cost of programming an FPL device.

Table 6-1 shows typical values for NRE costs and unit cost per device.

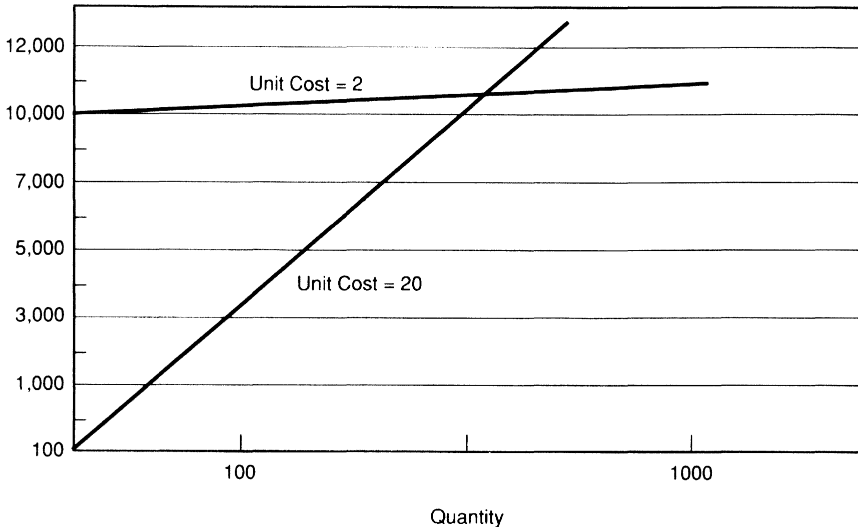
Table 6-1. Typical Alternative Costs

	NRE	Unit Cost
FPL	$100.00	$20.00
Gate Array	10,000.00	2.00

Figure 6-1 represents an estimate of the minimum production quantity at which FPL migration to ASIC is justified. Where production runs exceed this minimum quantity, long-term cost savings can be achieved by integrating one or more programmable devices into a single ASIC.

† Contributed by James T. O'Connor, P. E., ASIC Applications, Texas Instruments Incorporated.

Figure 6-1. Minimum Production Quantity for Migration



6.1.2 Basic Requirements of Migration Methodology

An automated migration methodology performs the following basic functions:

- Interprets the functionality of the FPL device
- Translates the FPL function into the target ASIC technology
- Outputs the translated function in an ASIC vendor's netlist format
- Develops a test program for manufacturing verification and control

For the methodology to interpret the functionality of the FPL device, the FPL netlist must be in a format that the translation software comprehends. Because FPL netlist formats are technology-specific and nonstandard, a general migration methodology requires preprocessing in order to convert the netlist to a standard description language.

Whichever language is used to describe the function, the FPL netlist must reference by name the FPL library components instantiated in the design. The translation software uses these names to search a reference library describing the function, ports, and timing of the FPL components in order to correctly interpret the function of the overall design.

An automated migration methodology correctly translates the FPL function into the target ASIC technology. The translation software uses a target library that describes the function, ports, and timing of the ASIC library components and facilitates the mapping of functions from FPL to ASIC devices.

To ensure efficient translation, the target library must contain components that are compatible with the FPL library. For example, if an FPL netlist includes a J-K flip-flop with preset and clear, the ASIC library must include a similar flip-flop.

In addition, the ASIC library must have a scan-cell equivalent function for each flip-flop in the FPL library to support automatic scan-chain insertion for test.

Finally, the ASIC library must offer I/O buffers that meet or exceed the electrical specifications of the FPL I/O buffers.

An automated migration methodology must output the translated function in the ASIC vendor's netlist format in order to perform ASIC layout and timing simulations. Although the logical functions of FPL and ASIC devices may be equivalent, technology differences and design techniques can result in different behavior or timing.

Prior to migration, assess whether the ASIC technology offers performance equal to or better than the FPL technology. After migration, perform postlayout simulations with back annotation in order to complete successful development of the ASIC.

The automated migration methodology must develop a test program for manufacturing verification and control. If the migration flow can insert test circuitry automatically and develop test vectors with high fault coverage, you will not need to develop a comprehensive set of test vectors that mimics the function of the device.

If critical path or output timing verification is required for manufacturing test, you may need to generate additional vectors using traditional ASIC development tools and add them to the test program.

6.1.3 Apply Good Digital Design Principles

The flexibility and low iteration costs of FPL devices encourage design practices that are unsuitable for automated, high-speed device testing in a manufacturing environment. If you plan to migrate FPL devices to ASICs, re-examine your design with manufacturing testability in mind. Design testability depends largely on the application of a few fundamental principles of good digital design.

The basic tenet in making a digital design testable is to ensure that internal nodes are controllable from the external input pins and observable at the external output pins.

The insertion of an internal scan chain for test requires that the following rules be observed in order to maximize controllability and observability.

- Make your design completely synchronous.

  Do not divide device clocks or combine them with other signals for use as the clock or data inputs to other registers or latches. Replace asynchronous paths, combinational feedback loops, and ripple counters with synchronous circuits.

- Make asynchronous preset and clear signals for registers and latches directly controllable from external pins.

  Asynchronous register signals generated from other registers make internal nodes uncontrollable and designs less testable.

- To implement a scan chain successfully for ATPG, ensure that all registers and latches are sensitive to the same edge or level of the clock.

  Conformance to this rule ensures that scan data will propagate reliably among scan registers.

Testability of a digital design is measured in terms of fault coverage. A fault is a node that cannot be changed from either a high (stuck-at-1) or low (stuck-at-0) state for a given input pattern. Fault coverage is defined as the percentage of observable faults over the total observable and unobservable faults. Follow the rules listed above to help maximize fault coverage and the quality of the manufacturing test vector set developed by an ATPG program.

Figure 6-2 and Figure 6-3 represent asynchronous and synchronous 3-bit counters that show the benefits of synchronous design techniques.Both circuits implement the function described in Table 6-2. The asynchronous counter in Figure 6-2 is relatively easier to design and compact than the synchronous counter in Figure 6-3, which requires more effort and gates.

Figure 6-2. Asynchronous 3-Bit Counter

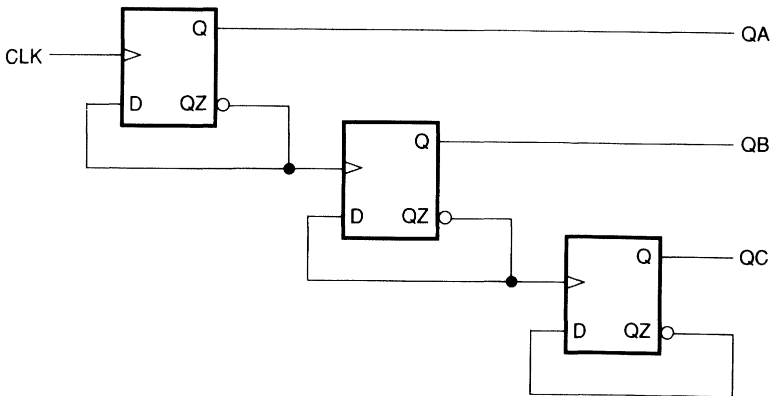


Figure 6-3. Synchronous 3-Bit Counter

CLK QB QC QA Q D QZ Q D QZ Q D QZ

Table 6-2. 3-Bit Counter State Table

Present State				Next State		
QC	QB	QA	CLK	QC	QB	QA
0	0	0	^	0	0	1
0	0	1	^	0	1	0
0	1	0	^	0	1	1
0	1	1	^	1	0	0
1	0	0	^	1	0	1
1	0	1	^	1	1	0
1	1	0	^	1	1	1
1	1	1	^	0	0	0

NOTE: ^ = positive-edge clock

Figure 6-4 and Figure 6-5 represent the same circuits after a scan chain is inserted. The CLK signal in the asynchronous 3-bit counter goes to only one flip-flop in the circuit; thus, only the first flip-flop may be replaced with a scan flip-flop (Figure 6-4). If the circuit is imbedded inside a design, none of the nodes associated with the nonscan flip-flops will be controllable or observable by a manufacturing test program.

Figure 6-4. Asynchronous 3-Bit Counter With Scan Chain

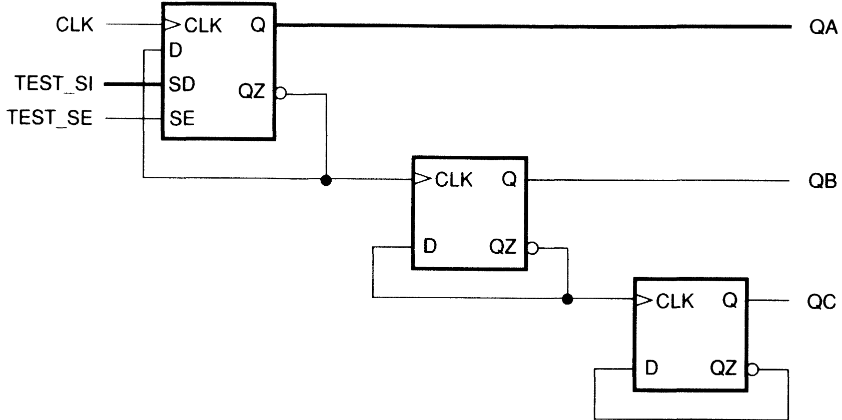
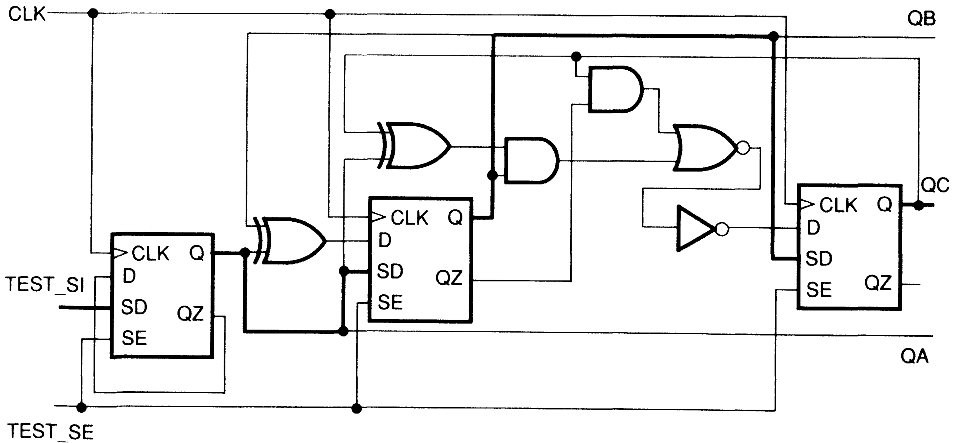


Figure 6-5. Synchronous 3-Bit Counter With Scan Chain



In addition, the timing of asynchronous speed paths from this circuit to an output pin should be different between the FPL version and the ASIC equivalent.

The flip-flops in the synchronous 3-bit counter are replaced with scan flip-flops in order to construct a serial-scan chain (Figure 6-5). All nodes in this circuit are controllable and observable in test mode as the scan vectors propagate from the *test_si* pin to the *QC* pin.

Table 6-3 summarizes the fault coverage resulting from these scan vectors for both asynchronous and synchronous designs. Not only is the fault coverage superior for the synchronous design, but timing differences between the FPL and ASIC versions of the circuit from asynchronous speed paths are eliminated.

Table 6-3. Fault Report for Counters

3-Bit Counter	Faults Observed	Faults Unobserved	Faults Total	Fault Coverage
Async.	8	20	28	29
Sync.	5	20	52	100

As long as the critical path in the ASIC is as fast or faster than the FPL device, the behavior of both ASIC and FPL devices will be the same.

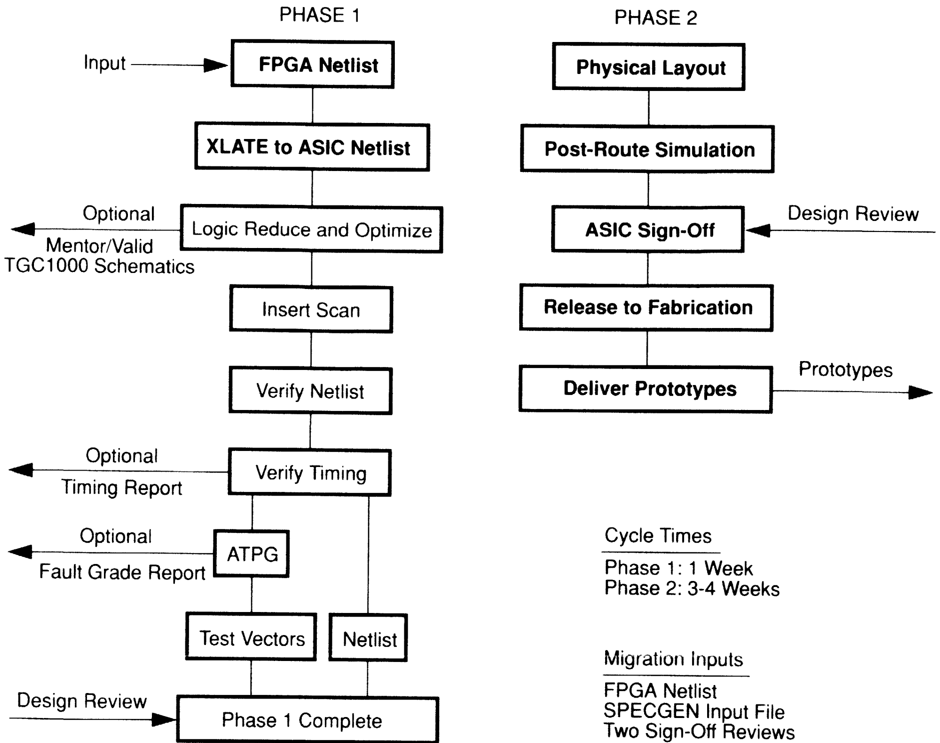
6.1.4 FPGA Migration to ASIC

Figure 6-6 illustrates a TI FPGA-to-ASIC flow. Phase I represents the technology translation and test vector generation steps of the migration flow. Phase II represents typical methodologies for layout, manufacturing, and testing of an ASIC device. Nothing in phase II is modified in order to complete a successful migration.

The flow originally supported TI and Actel FPGA migration; however, after additional development, interfaces were added to support the migration of Xilinx X2000, X3000, and X4000 FPGAs, as well as Altera EP and EPM series EPLDs. Both a 1.2–μm gate array library and 1.0–μm standard cell library are available as target ASIC technologies.

The migration flow has two inputs: the first input is the FPGA netlist, either an `.adl` netlist for TI FPGAs, an `.xnf` netlist for Xilinx FPGAs, or a `.rpt` file for Altera EPLDs. These netlists may come from any FPGA vendor-supported platform.

Figure 6-6. FPGA-to-ASIC Migration Flow



The second input is a text file in electronic form filled in by the FPGA designer. The text file specifies the pinout, I/O buffer selection, and electrical or timing characteristics of the final ASIC.

Prior to technology translation, the FPGA netlist itself is translated into a standard hardware description language, preserving all of the primitive cell calls and interconnectivity of the original FPGA. This standard HDL description is then read and translated into the equivalent ASIC function.

The translation software automatically compares ASIC and FPGA functions to verify accurate translation and has the ability to optimize the ASIC design for area and speed.

The translation software can produce Mentor or Valid workstation schematics and continue ASIC development using traditional methods of simulation and test vector development.

Another program in the migration flow can recompile the ASIC design, automatically insert an internal scan chain, and develop scan test vectors. The result is a significant productivity enhancement.

The scan chain is constructed by inserting a multiplexer at the data input of existing flip-flops in the design. A global scan-enable signal drives all multiplexer enable inputs; the scan data propagates serially from the output of one scan-cell flip-flop to the scan-data input of the next.

In addition to the multiplexed flip-flop scan methodology, the software will implement other scan methodologies under development for FPGA migration.

With the scan chain in place, the program automatically develops test vectors, compacts them to minimize test time per device, and generates a fault coverage report. If the design conforms to the digital design practices described in Section 6.1.3, the program will produce high quality test vectors with fault coverage typically greater than 90 percent. In this case, the resulting test program is generally suitable for manufacturing test.

No other test vector development is required, unless additional vectors are needed to test critical speed paths. If the fault coverage is low, the program will generate a report that identifies all untestable nodes and enables you to modify the logic to improve the fault coverage.

At the completion of phase I, the new ASIC circuit description and test vectors are written in TI internal standard formats and sent on for prototype layout, test, and manufacturing.

The execution of phase I is accomplished entirely with specialized software, which greatly improves productivity. Although actual execution time for phase 1 is design-dependent, typical times range from two to five man-days with no design iterations. This compares favorably to migration methods employing traditional ASIC tools for schematic capture, simulation and test vector development that typically require on the order of two to five man-months.

6.2 FPGA to ASIC Migration Options†

Using TI FPGA devices to prototype your new design or run a first series will help you improve your time to market. For high volume production, TI ASIC products offer the following benefits:

- ❑ Lower chip price
- ❑ No need for programming
- ❑ Tested devices
- ❑ Performance improvement
- ❑ Ultimately higher density by integrating several FPGAs to one ASIC

The following options are available to replace an FPGA device with a customized product:

- ❑ Hardwired FPGAs
- ❑ Preprogrammed FPGAs
- ❑ Migration to ASIC

6.2.1 Hardwired FPGAs

Hardwired FPGAs offer a possible solution for small FPGA designs; however, their chip-price reduction factor of 2:4 is insufficient for larger, more expensive designs. In addition, the requirement to produce a new mask set increases the possibility of a long lead time. The narrow application of HALs have not proven to be a successful option.

6.2.2 Preprogrammed FPGAs

TI preprogrammed FPGAs offer an effective solution for small volume products. Similar to TI preprogrammed PLDs, FPGAs are programmed to customer specification and tested in the factory. Postprogramming testing improves the quality of the final device by order of magnitude.

6.2.3 FPGA Migration to ASIC

For high volume products, migration of your FPGA devices to ASIC offers all the benefits listed in Section 6.2, whether you are converting one FPGA to one pin-compatible gate array or integrating several FPGAs to one gate array or standard cell.

There are three basic approaches to migration:

1) Schematic translation
2) Netlist translation
3) Logic synthesis using a high-level hardware description language

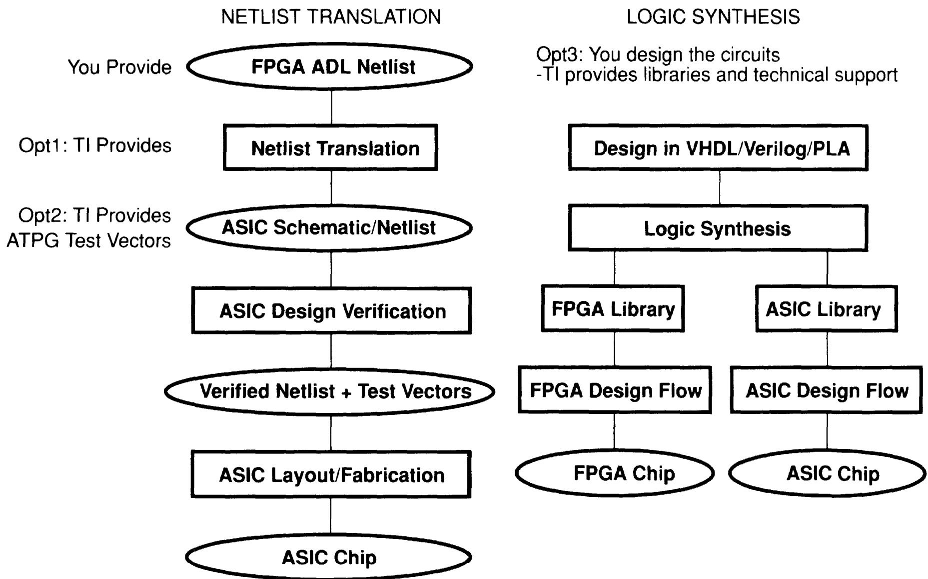
† Contributed by Dung Tu, Technical Marketing, Texas Instruments Deutschland GmbH.

Schematic translation provides one-to-one manual or software replacement of each component on the original schematic with a component from the new technology library. Because FPGA and ASIC architectures differ, however, schematic translation does not allow efficient use of the ASIC.

TI supports both netlist translation and logic synthesis flow (Figure 6-7). Netlist translation maps an FPGA netlist to an ASIC netlist, which can be optimized for ASIC.

Logic synthesis provides a technology-independent design that can be described using a high-level description language and later mapped to the required silicon technologies using the appropriate libraries. Of the two approaches, logic synthesis is the most universal, providing not only migration from FPGA to ASIC but also from ASIC to FPGA.

Figure 6-7. TI FPGA-to-ASIC Migration



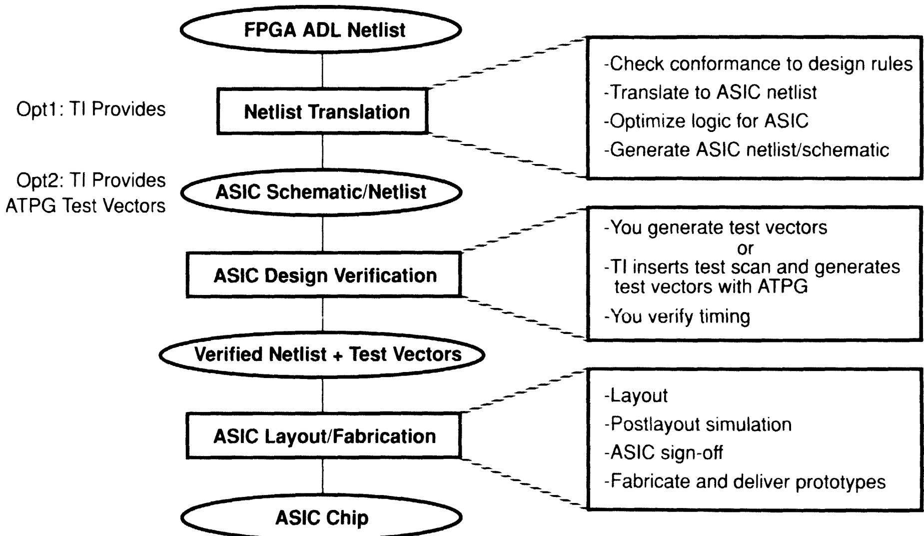
NOTE: For multiple FPGA migration, you must create one ADL netlist.

6.2.3.1 Netlist Translation

Starting from an FPGA netlist in ADL format, the first step in the design flow is to ensure that the netlist conforms to design rules for correct syntax, good testability, and correct pinout for the ASIC. The FPGA netlist is then translated and optimized to an ASIC netlist using TI software and the Synopsys logic synthesis tool. The TI ASIC can be either a TGC1000 CMOS gate array or TSC700 standard cell product.

Figure 6-8 shows the service options available from TI. Option 1 provides a pure netlist translation. Option 2 provides netlist translation and ASIC test vectors generated by an ATPG (Synopsys test compiler).

Figure 6-8. Netlist Translation Flow



Although ATPG requires sound design-for-test practice, you are released from the requirements of manual test vector generation, which is the most time-consuming and least understood part of the ASIC design process, consuming up to 40 percent of the total design time.

Test vectors with 100 percent test coverage are easy to develop for combinational circuits. Testing sequential circuits is more complicated. Without a test-scan technique, a sequential circuit as shown in Figure 6-9 is difficult to test because the present state of the sequential cells depends on the previous state. Scan design requires replacement of the sequential elements with a modified, scanable version that also performs a serial shift function.

The following types of scan cells are available:

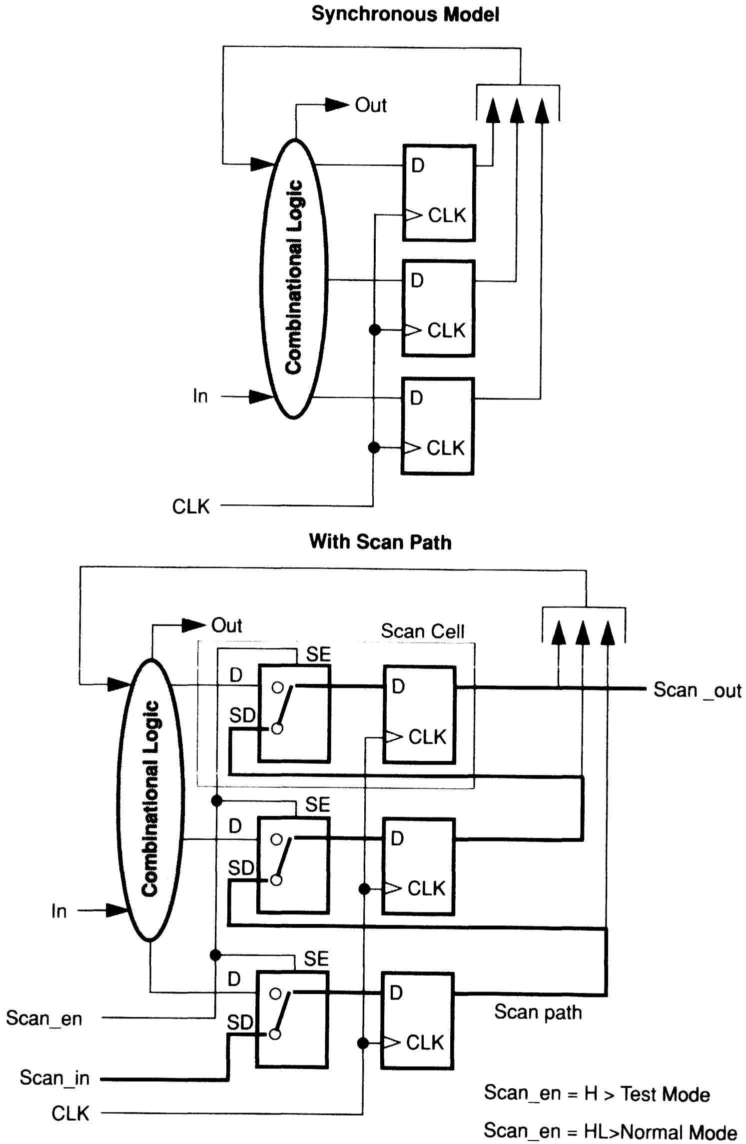
1) Multiplexed flip-flop
2) Clocked scan
3) Level-sensitive scan design (LSSD)

Figure 6-9 illustrates multiplexed flip-flops used as scan cells. The circuit has two additional inputs, *Scan_enable*, *Scan_in*, and one additional output, *Scan_out*. Depending on the *Scan_enable* input, the multiplexed flip-flops work either in the normal mode or test mode.

In the test mode the flip-flops are connected by the scan path to a shift register. The flip-flops can be tested by shifting a binary pattern into *Scan_in* and monitoring the output at *Scan_out*.

Another popular scan cell is a master-slave flip-flop, which is a double-latch LSSD hard macro in the TGC1000 gate array and TSC700 standard cell libraries.

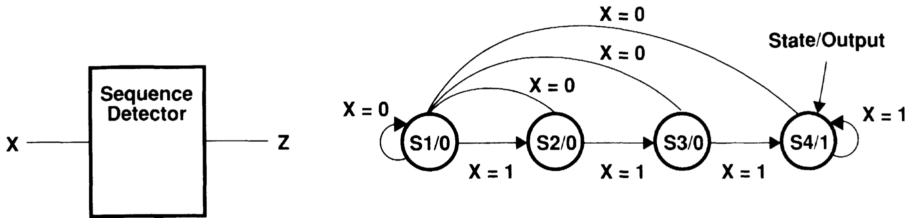
Figure 6-9. Scan Cell and Scan Path



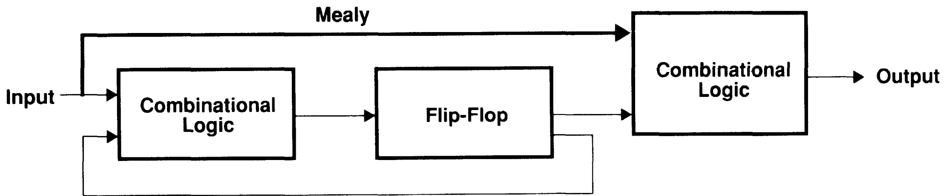
6.2.4 Migration Example

Figure 6-10 shows a simple state machine design example in the form of a sequence detector. The circuit monitors an input signal *X* and outputs a signal *Z=1* whenever *X=1* is true for three successive clock cycles.

Figure 6-10. Sequence Detector State Diagram



- **The sequence detector monitors an input signal *X* and outputs a signal *Z=1* when *X=1* for three successive clock cycles.**



- **The state machine is implemented as a Mealy machine; e.g. the output depends not only on the state but also on the inputs.**

Figure 6-11 shows the FPGA schematic with back-annotation simulation using Viewlogic.

Figure 6-11. Sequence Detector FPGA Schematic

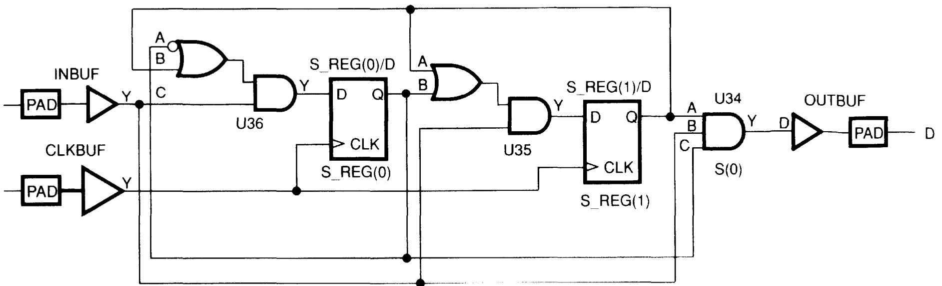


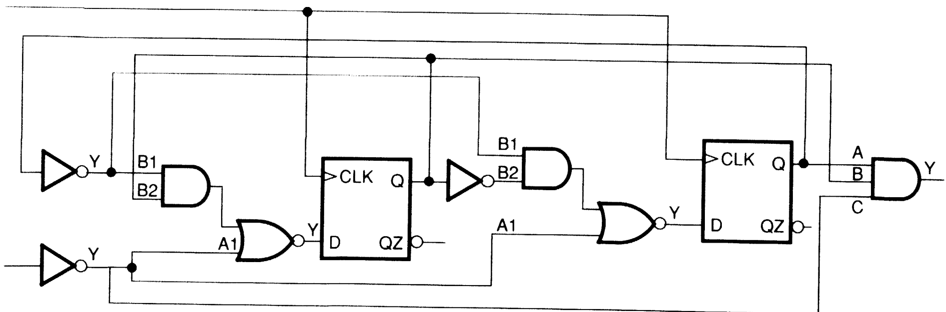
Figure 6-12 depicts the ADL netlist.

Figure 6-12. ADL Netlist

```
DEF MEASEQ; CLK, X, Z.
USE ADLIB:DF1;S_REG[1].
USE ADLIB:AND3;U34.
USE ADLIB:DF1;S_REG[0].
USE ADLIB:OA1;U35.
USE ADLIB:OA1A;U36.
USE ADLIB:CLKBUF;$1|17.
USE ADLIB:OUTBUF;$1|18.
USE ADLIB:INBUF;$1|16.
NET CLK;CLK,$1|17:PAD.
NET X; X,$1|16:PAD.
NET Z; Z,$1|16:PAD.
NET $1N10;U34:B, U36:C, U35:C, $1|16:Y.
NET $1N11; S_REG[0]:CLK, S_REG[1]:CLK, $1|17:Y.
NET $1N28; $1|18:D, U34:Y.
NET S0; U36:A, U35:B, U34:C, S_REG[0]:Q.
NET S0; U34:A, U36:B, U35:C, S_REG[1]:Q.
NET S_REG0/D; S_REG[0]:D, U36:Y.
NET S_REG1/D; S_REG[1]:D, U35:Y.
END.
```

After the netlist translation, a schematic can be generated without test scan for the same design using TGC1000 gate arrays (Figure 6-13).

Figure 6-13. TGC1000 Schematic Without Test Scan



The same circuit with test scan is shown in Figure 6-14. The D flip-flops are replaced by the multiplexed flip-flops.

Figure 6-14. TGC1000 Schematic With Test Scan

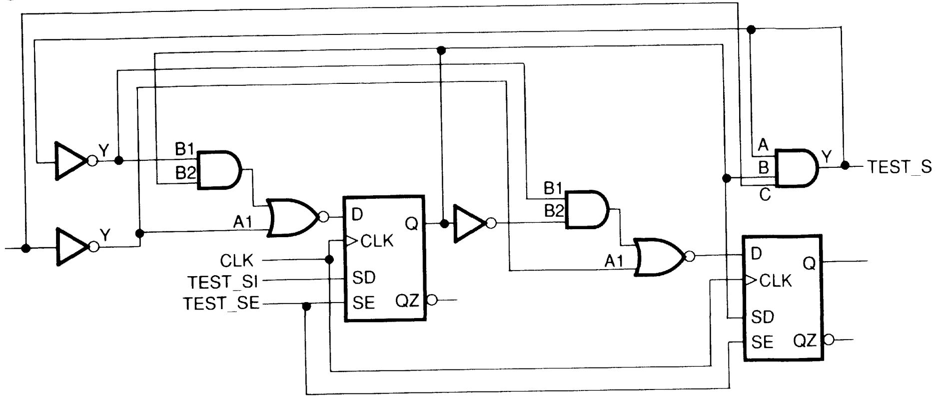


Figure 6-15 shows the test vectors in TI TDL format. The newest version of the Synopsys test compiler includes an option to generate the TDL format.

Figure 6-15. TI TDL Format

```
(*$ DATE 9/18/91 12:43                                          *)

(*$ TESTER: ACTV *)
(*$ TEST TYPE: FUNCTIONAL LIBRARY/TIMING TGC1000 COMMERCIAL *)
(*$                                                             *)
CONNECT P, VAR=(SCANEN,CLK,SCANIN,SCANOUT,X,Z,CLR,TESTB),
DEFPIN=(IN 3,OUT,in,OUT,IN 2))
(*$ PERIOD=1000/0;*)
CLOCK VAR=CLK,
 PATTERN=010, HOLD0= 300,HOLD1= 300;
DELAY VAR=TESTB,
 OFFSET= 0;
DELAY VAR=CLR,
 OFFSET= 0;
DELAY VAR=X,
 OFFSET= 0;
DELAY VAR=SCANIN,
 OFFSET= 0;
DELAY VAR=SCANIN,
 OFFSET= 0;
STROBE VAR=Z,
 OFFSET= 900;
STROBE VAR=SCANOUT,
 OFFSET= 900;
(*$   *)
(*$  SCSSXZCT*)
(*$  CLCC LE*)
(*$  AKAA RS*)
(*$  N NN T*)
(*$  E IO B*)
(*$  N NNU *)
(*$    T *)
```

```
        SETR P:=T'YLYMYMYY';(*& 0*)
        SETR P:=T'HCHMHMYL';(*& 1*)
        SETR P:=T'HCL1HMYL';(*& 2*)
        SETR P:=T'HCY0HMYL';(*& 3*)
        SETR P:=T'YLYMYMYY';(*& 4*)
        SETR P:=T'HCHMHMHL';(*& 5*)
        SETR P:=T'HCHMHMHL';(*& 6*)
        SETR P:=T'LLHMH1HL';(*& 7*)
        SETR P:=T'LCH1HMHL';(*& 8*)
        SETR P:=T'HCH1HMHL';(*& 9*)
        SETR P:=T'HCLMHMHL';(*& 10*)
        SETR P:=T'LLLMH0HL';(*& 11*)
        SETR P:=T'LCL1HMHL';(*& 12*)
        SETR P:=T'HCL1HMHL';(*& 13*)
        SETR P:=T'HCLMHMHL';(*& 14*)
        SETR P:=T'LLHMH0HL';(*& 15*)
        SETR P:=T'LCH0HMHL';(*& 16*)
        SETR P:=T'HCH1HMHL';(*& 17*)
        SETR P:=T'HCHMHMHL';(*& 18*)
        SETR P:=T'LLHML0HL';(*& 19*)
        SETR P:=T'LCH0LMHL';(*& 20*)
        SETR P:=T'HCL0LMHL';(*& 21*)
        SETR P:=T'HCHMLMHL';(*& 22*)
        SETR P:=T'LLHMH0HL';(*& 23*)
        SETR P:=T'LCH1HMHL';(*& 24*)
        SETR P:=T'HCY0HMHL';(*& 25*)
        END;
($* The following input(s) did not transition to L:*);
        ($* CLR *);
($* The following input(s) did not transition to H:*);
        ($* CLK *);
        ($* TESTB                                    *);
```

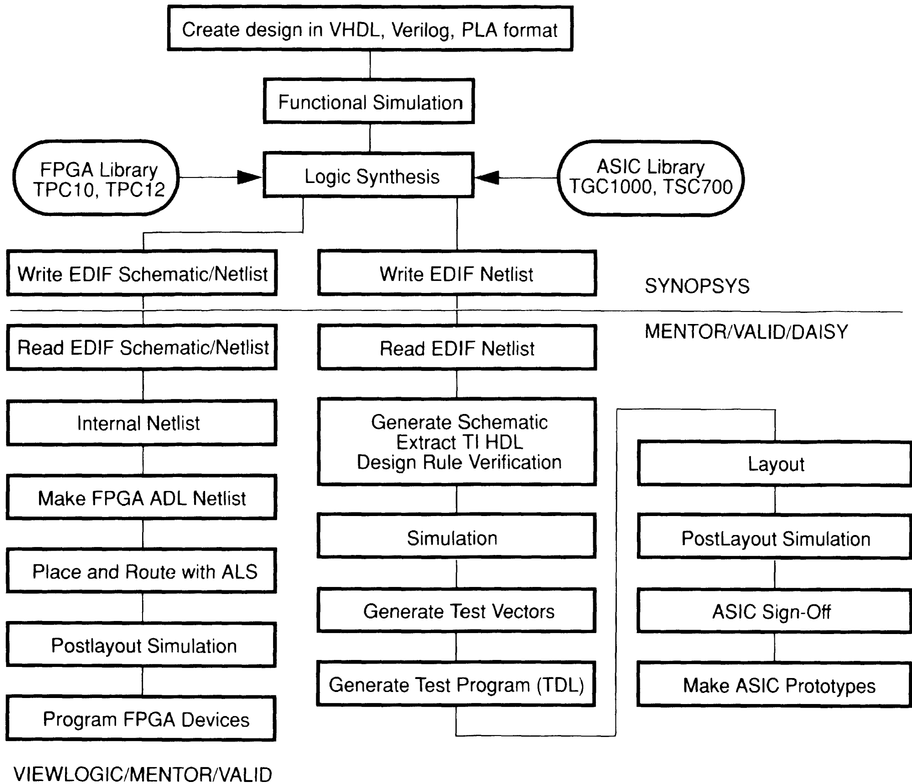
26 test vectors are needed to test 8 gates!

6.2.5 Logic Synthesis Migration Flow

Logic synthesis implicitly allows migration by virtue of its technology-independant design methodology. Figure 6-16 shows the migration flow using logic synthesis.

A high-level language such as VHDL or Cadence Verilog is used to describe your design, the functionality of which is verified using a VHDL or Verilog simulator. The design is synthesized with a logic synthesis tool, such as the Synopsys design compiler. Depending on the required technologies, either an FPGA library or ASIC library is used for the synthesis. The result is written out as an EDIF netlist or EDIF schematic.

Figure 6-16. Logic Synthesis Migration Flow



An EDIF interface is required to convert the EDIF netlist to the correct format for the design platform you use to develop the FPGA or ASIC. For FPGA, use the ADL netlist generation utility included in the FPGA development kit to convert the internal netlist to an ADL netlist. The ADL netlist is used by TI-ALS to place and route the design. After you complete a successful postlayout simulation, program your FPGA chips. Use the EDIF netlist as the starting point for the ASIC design flow.

Figure 6-17 depicts the VHDL code for the sequence detector and the script file for the Synopsys design compiler. For FPGA, specify FPGA libraries as the link library, target library, and symbol library. For ASIC, specify the TGC1000 gate array library or TSC700 standard cell library as appropriate.

Figure 6-17. VHDL Code and Script File

VHDL Source

```
entity MEASEQ is
 port(X, CLK, : in BIT)
      Z     :out BIT);
end;
architecture behavior of MEASEQ is
 type STATE_TYPE is (S1,S2,S3,S4);
 signal S, SNEXT: STATE_TYPE;
begin
 COMBIN: process
 begin
  case S is
   when S1 =>
    if X = '1' then Z = '0';  SNEXT <= S2;
    else            Z = '0';  SNEXT <= S1;
    end if;
 when S2 =>
    if X = '1' then Z = '0';  SNEXT <= S3;
    else            Z = '0';  SNEXT <= S1;
    end if;
when S3 =>
    if X = '1' then Z = '0';  SNEXT <= S4;
    else            Z = '0';  SNEXT <= S1;
    end if;
when S4 =>
    if X = '1' then Z = '0';  SNEXT <= S4;

    else            Z = '0';  SNEXT <= S1;
    end case;
  end process;
  SYNCH: process
  begin
    wait until CLK' event and CLK ='1';
     S <=SNEXT;
   end process;
  end BEHAVIOR;
```

Script File

```
/***Design check***/
designer        ="Dung Tu";
company         = "Texas Instruments";
search_path     ={./winl/lib/ti/fpga}
link_library   =tpc10.db
target_library  tpc10.db

symbol library =tpc10.sdb
read -f vhdl measeq.vhdl
current_design = MEASEQ
write -h -t db -hierarchy -o measeq.db
check_design > chkdsn.rpt
report_design > measeq.rpt
report_design > measeq.rpt

/***Constraint: Optimize for area***/
max_area 0.0
compile

write -h -f db -hierarchy -o measeq_opt.db
report_area -cell -timing > measeq.rpt
free -all

/***Write  EDIF  schematic  for  export  to
 Valid, Mentor or Viewlogic***/
read -f db measeq_opt.db
current_design = MEASEQ
create_schematic -hierarchy -size B
write -h -f edif -o measeq.edif
exit
```

The key benefit of the logic synthesis flow is that the design source code remains the same. The logic synthesis tool takes care of the implementation on the gate level. Compare the gate array schematic for the sequence detector in Figure 6-13 with the FPGA schematic in Figure 6-11 to see that the TGC1000 hard macros and the FPGA soft macros are not the same. Logic synthesis releases you from these kinds of details.

The FPGA-to-ASIC migration path is not the same as the ASIC-to-FPGA path. When you design an ASIC you normally include scan cells and scan path in the source code. If you use the same code for FPGA implementation, the result will be a big overhead because of the test cells. A large ASIC design may require several FPGAs, requiring partitioning of the design.

As supplier, TI supports migration between FPGA and ASIC as well as netlist translation and logic synthesis. Although netlist translation converts an existing FPGA design to an ASIC, logic synthesis is clearly the design flow of choice for new designs implemented in different technologies.

6.3 Migration Design Rules for Testability[†]

FPGA is an excellent tool for testing new design concepts and for marketing a product quickly. The ability to quickly and easily migrate to a low-cost gate array increases the appeal of both FPGA and gate array products.

To be viable, migration should require as little effort as possible. ATPG supports this concept by reducing the time needed for test vector generation, which takes up a large percentage of engineering time. Although FPGA designs include glue logic from different areas of a PC board where testing is difficult, these problems can be minimized by following basic design-for-testability rules.

Understanding design-for-testability rules starts with the migration flow. Figure 6-18 shows the basic elements of the migration flow used by TI, which receives a netlist from you for translation into a standard Verilog format.

The netlist is mapped or translated into the target ASIC library. Next, the design is optimized for area or timing. After optimization, the circuit is checked for test rule violations, and scan flip-flops are inserted and connected into a scan chain.

The next step in the flow is to generate test vectors and a fault coverage report. The vectors are translated into the TI TDL to be used for production test. Once the netlist is finished, either a Mentor or Valid schematic database can be generated and handed off to you for system timing verification.

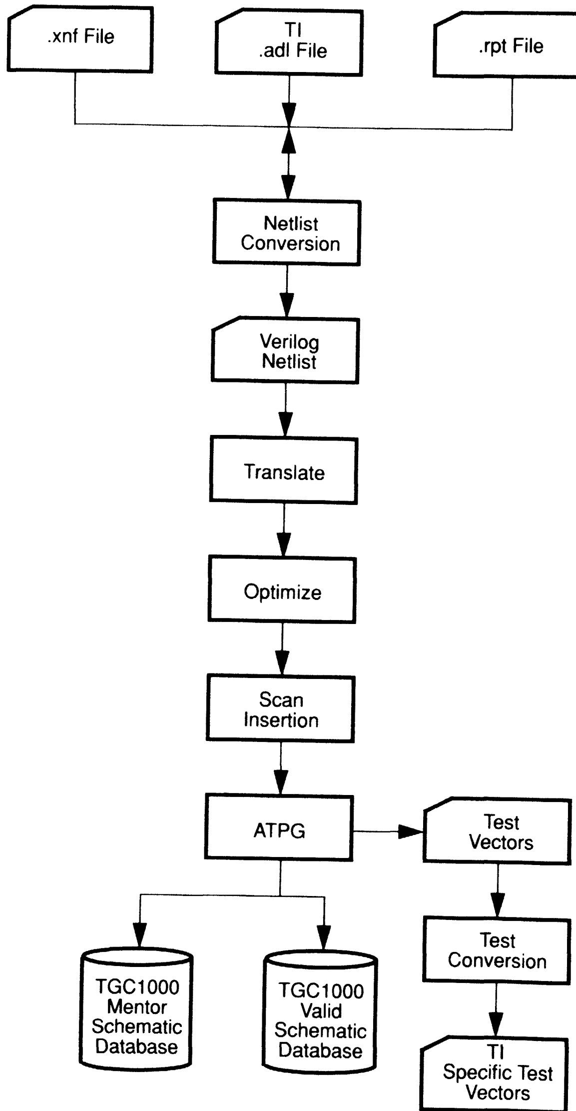
TI uses a multiplexed flip-flop scan methodology for scan implementation. Each flip-flop in the scan chain is replaced with an equivalent flip-flop and multiplexer. When the scan chain is connected, the *q/scan_out* of one flop is routed to the *scan_in* of the next. The *scan_enable* is routed globally to all flip-flops in the scan chain.

The methodology offers the following advantages:

- Although minimum of one additional signal pin (*scan_enable*) must be added, the addition of *scan_in* and *scan_out* pins may not be necessary if multiplexed with design signals.
- Low area overhead. The multiplexed flip-flop is normally no more than 30 percent larger than the basic flip-flop. The only signal that must be globally routed is *scan_enable*.
- Easily implemented using virtually any library containing a flip-flop and multiplexer.

[†] Contributed by Patrick D. Eyres, ASIC Applications, Texas Instruments Incorporated.

Figure 6-18. TI Migration Flow



The methodology also includes the following trade-offs:

- Additional setup time because of the addition of a multiplexer in front of every flip-flop.
- It is unsuited for designs containing latches, which cannot be included in the scan chain and can only be tested if made transparent.

To achieve high fault coverage, a flip-flop must be included in the scan chain, or the ATPG software will consider it to be a black box. In that case, no node between the black box and any other flip-flop will be tested.

Unfortunately, including every flip-flop in the scan chain is not always possible because of the design rules imposed by ATPG software. It is thus essential to understand and follow the rules.

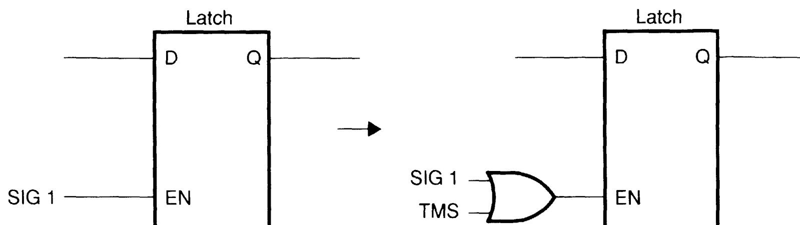
A common technique is used to correct most problems. ATPG software allows the use of the Test-Mode-Select (TMS) pin, which can be asserted during vector generation, causing the circuit to go into test mode. In test mode, the design contains fewer rule violations and the scan chain includes more flip-flops.

6.3.1 Latches

The multiplexed flip-flop methodology prohibits latches in the scan chain. The preferred solution is to replace all latches with flip-flops. If this is not possible, the following alternatives are available.

- Control the enable pin from a top-level signal pin (the objective of any design change dealing with latches), which will cause the ATPG software to make the latches transparent during testing.
- Add the circuitry as shown in Figure 6-19. In this example, SIG1 is the normal enable for the latch. During test, TMS is held to a logic 1 and the latch is transparent. Fault coverage on the latch itself is still relatively poor. When the latch is made transparent, several nodes will not be toggled.

Figure 6-19. Example of Circuitry Used as Alternative to Replacing Flip-Flops

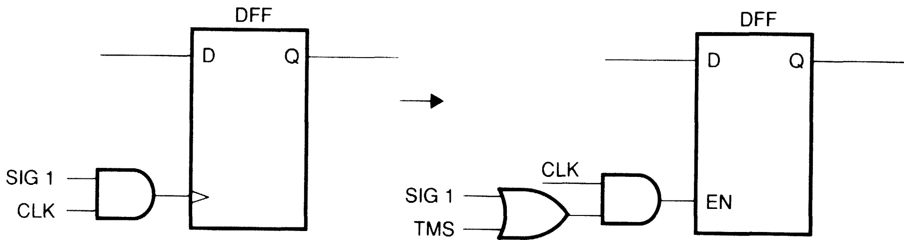


6.3.2 Gated Clocks

The use of gated clocks in your design rarely consitutes good design practice and should be avoided, if possible. Figure 6-20 shows an example of a gated clock and a possible solution. The clock of any flip-flop in the scan chain must be fully controllable from the top-level signal pins. If SIG1 in the example is generated internally, CLK may not reach DFF during scan, causing DFF to be excluded from the scan chain.

As the solution shows, TMS is held to a logic 1 and DFF is clocked. Adding test circuitry also adds potential faults. The OR gate and half of the AND gate will not be tested. If the flip-flop is added to the scan chain, the vectors will catch faults between this flip-flop and any others.

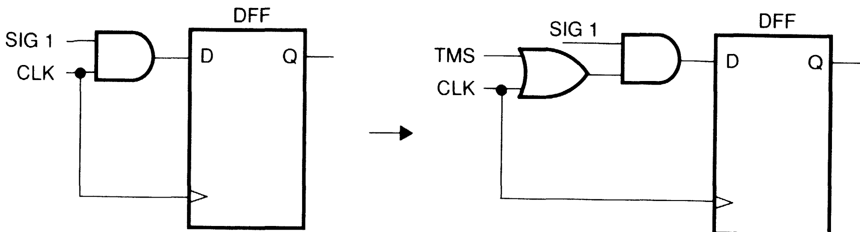
Figure 6-20. Example of Solution to Gated Clock



6.3.3 Clock Signals Used for Data

Figure 6-21 shows another example of poor design practice. A circuit with a clock going to the data input of a flip-flop can cause possible setup violations. The problem occurs when the design has two independent clocks. The ATPG software hooks all flip-flops, regardless of which clock they use, into one scan chain. All clocks in the design are then clocked together during test. If one clock goes to the data input of a flip-flop that is clocked by another, a setup violation may occur. Once again, using the TMS pin can ensure that the CLK signal does not reach the data pin during test mode.

Figure 6-21. Example of Poor Design Practice Causing Possible Setup Violations



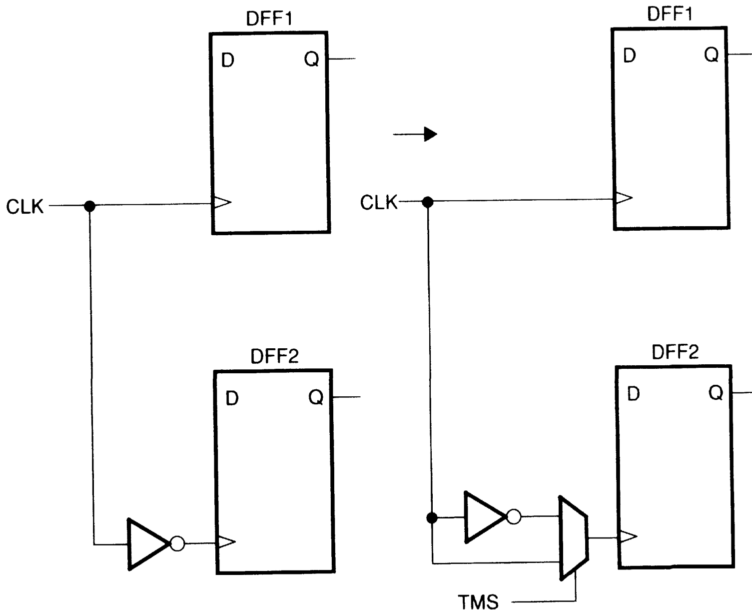
6.3.4 Using Both Edges of a Clock

The ATPG software assumes that all flip-flops are clocked from the same edge of the clock. If a flip-flop clocked on the falling edge of the clock appears in the scan chain after a flip-flop clocked on the rising edge, scan data will be clocked into the two flip-flops in one clock cycle: the first on the rising edge, the second on the falling edge.

If three flip-flops are in the chain, the software will apply a pattern of values in three clocks, causing the the falling edge flip-flop to corrupt the chain. If you have two different clocks, you can clock one on the rising edge and one on the falling; however, for any one clock you can only use one edge.

Figure 6-22 shows an example of a multiplexer used in front of the flip-flop clocked on the falling edge. During test, TMS will select the rising edge instead.

Figure 6-22. Example of Solution to Using Both Edges of Clock

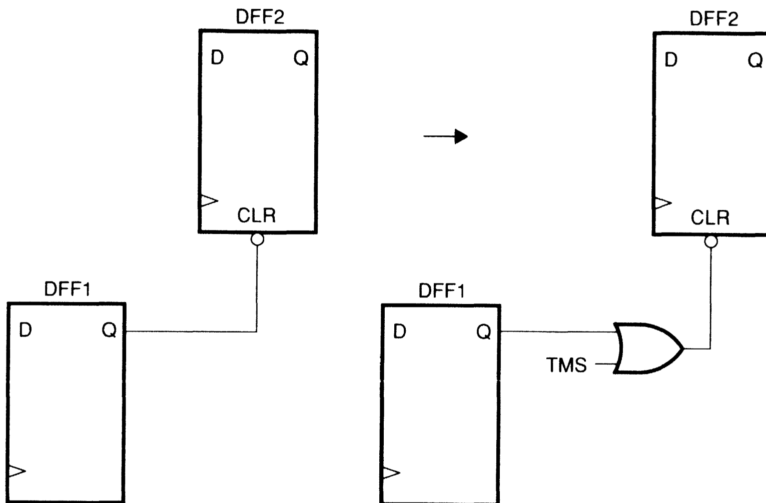


6.3.5 Uncontrollable Asynchronous Pins

To put a flip-flop in the scan chain, the software must be able to control the asynchronous signals on the flip-flops. A scan pattern should not clear or preset another flip-flop in the scan chain. The preferred design method is to make all asynchronous pins controllable from a top-level signal.

Figure 6-23 shows an example. If a logical 0 is scanned into DFF1, DFF2 will be cleared, breaking the scan chain. By inserting an OR gate, the path is effectively cut during test mode.

Figure 6-23. Example of Solution for Uncontrollable Asynchronous Pins

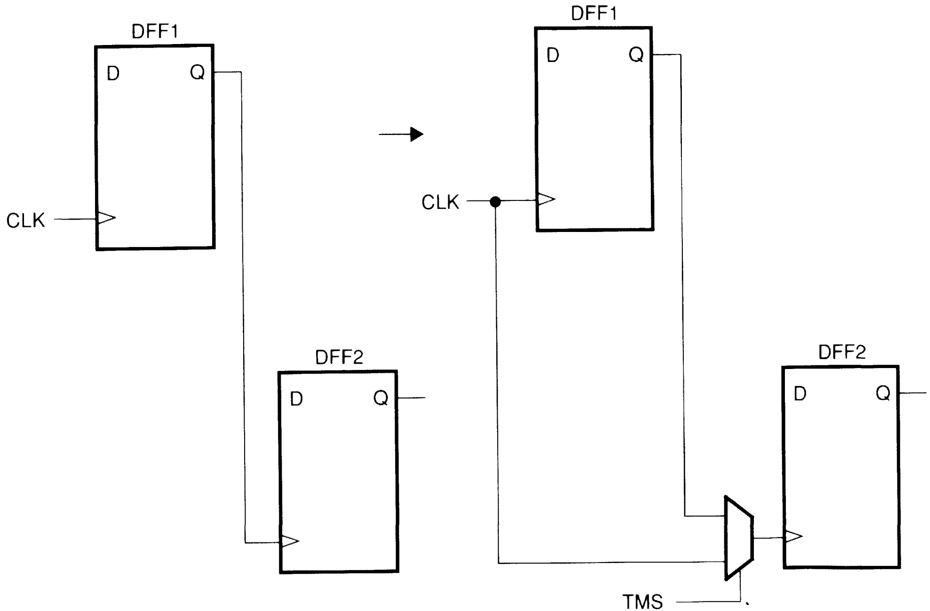


6.3.6 Q OUT Used to Clock a Flip-Flop

Using an internally generated signal to clock a flip-flop will cause a rules violation. The clock of the second flip-flop will be uncontrollable and therefore excluded from the scan chain.

All flip-flops should be clocked from a single system clock. A multiplexer can be added to the circuit, as shown in Figure 6-24. During test mode the multiplexer will select the system clock. DFF2 will then be included in the scan chain.

Figure 6-24. Example of Q OUT Used to Clock Flip-Flop



The techniques presented here are not exclusive. In each case, you should first attempt to redesign circuits that violate ATPG rules. If this cannot be done, the TMS pin or other similar techniques can be effectively used.

These techniques are more than good design practice. By adhering to all design rules, you will need only to hand off an FPGA netlist. Turnaround time from handoff to gate-array schematic can be as fast as six days. The design will have all the needed production test vectors with no additional work required from you.

6.4 Concurrent FPGA and ASIC Design Using HDL†

Designing ASICs and FPGAs in parallel using hardware description languages and logic synthesis tools allows rapid prototyping along with the possibility of design verification in hardware prior to the creation of masks, thus reducing the risks associated with ASIC design. While an ASIC is being produced, additional hardware and software; e.g., microcontroller applications, can be developed and tested. This section discusses recommendations for this design methodology.

FPGA programmability and architecture has the following disadvantages when compared to ASIC:

- Larger gate delays
- Larger wire delays
- Limited maximum system clock frequency
- Limited gate count
- Limited number of pads and types of pad cells (slew rate, Schmitt trigger, CMOS, TTL, driving strength)
- Limited number of on-chip interconnection wires (routing sources)
- Limited number of sequential cells
- No oscillators on chip possible
- No analog modules on chip possible (mixed-mode design)

The following guidelines are recommended for parallel designing:

- Plan your design management well.
- Be consistent throughout your design (hierarchy, modules, signal names, etc.)
- Combine ASIC- and FPGA-specific modules in a common source code and select by conditional compiling (e.g., clock oscillators, NAND tree, and test and scanpath cells are not necessary on FPGAs).
- Choose common sequential cells on ASIC and FPGA libraries.
- Make your design as fully synchronous as possible.
- Use a common clock tree (e.g., built-in FPGA clock distribution network).
- Avoid glitches and race conditions.
- Avoid using gated clocks, which are hard to implement on FPGAs and waste routing resources).
- Create all modules within FPGA limits (e.g., gate count and number of pads).
- Avoid bus-intensive architectures, which are hard to route on FPGAs.
- Ease design partitioning.
  - Create a separate pad level containing only pad cells.
  - Create a separate interconnection level below the pad level that contains only module instanciations and interconnections but no primitive cells.

† Contributed by Peter Heusingen and Norbert Schuhmann, Fraunhofer Institute for Integrated Circuits.

If your design will not fit into one FPGA, partitioning will be necessary. The following partitioning guidelines are recommended:

- Combine the pad and interconnection level to a new interconnection level.
- Distribute modules and related pads below the interconnection level to different FPGAs (monitoring gate count and pad limitations).
- Insert interconnection pads for inter-FPGA connections (watch for additional delays by interconnections and interconnection pads).
- Implement the master clock net as an interchip clock tree.
- Implement the internal clock by the built-in distributed clock.

Chapter 7

FPGA Programming and Test

This chapter discusses FPGA manufacturing considerations and programming options, testing features (including FPGA-to-ASIC migration using Synopsys design and test compilers), design-for-testability techniques used in factory testing, and fuse verification testing during the Activate process. In addition, FPGA copy protection is described.

7.1 Programming FPGAs - Considerations and Options[†]

TI FPGA products offer you a flexible design tool to reduce cycle time, package counts, and NRE costs. The desktop design and programming capability helps you deliver your new design to market in less time; however, the quick-turn capability you gain creates unique manufacturing challenges as your design moves from prototype to production.

TI FPGA devices must be individually programmed prior to use, either by an in-house programming capability or outside programming sources. As you consider whether or not to establish an in-house programming capability, you will be confronted with these key manufacturing issues and programming options:

- ❑ Programming time and throughput
- ❑ Maintaining lead quality of surface mount components
- ❑ Handling moisture-sensitive plastic surface mount components (PSMCs)
- ❑ Manufacturing efficiency

Recognizing the complexity involved in establishing an efficient programming process, TI offers factory-programmed FPGAs (P-FPGAs) to help minimize your in-house manufacturing requirements.

Choosing TI to program your FPGAs lends you the support of TI's worldwide resources, industry leading manufacturing expertise, and quality standards. Your devices are delivered fully programmed, symbolized, and ready to meet your volume production requirements.

Another programming option is to use the facilities and services provided by TI-authorized distributors and representatives, many with TI-certified facilities capable of fulfilling your volume production requirements.

7.1.1 Programming Time and Throughput

TI FPGA programming time is a function of antifuse technology and fuse density. Antifuses require multiple programming pulses to form the electrical connection between logic modules. Depending on the device type, 3,000 to 20,000 antifuses are typically programmed.

Figure 7-1 shows a typical per-unit programming time for TI FPGAs using the TI Activator programming unit.

[†] Contributed by Jerry Hughes, Manager, Factory Programmed Parts, Texas Instruments Incorporated.

Excessive device programming time is not a critical issue during design verification and system prototyping; however, after a design is ready for production, an efficient programming operation must be implemented to minimize cycle time.

Because programming cycle time increases proportionally to the quantity of units programmed, high-volume production requirements place a heavy burden on programming operations.

Figure 7-1. FPGA Per-Unit Programming Time

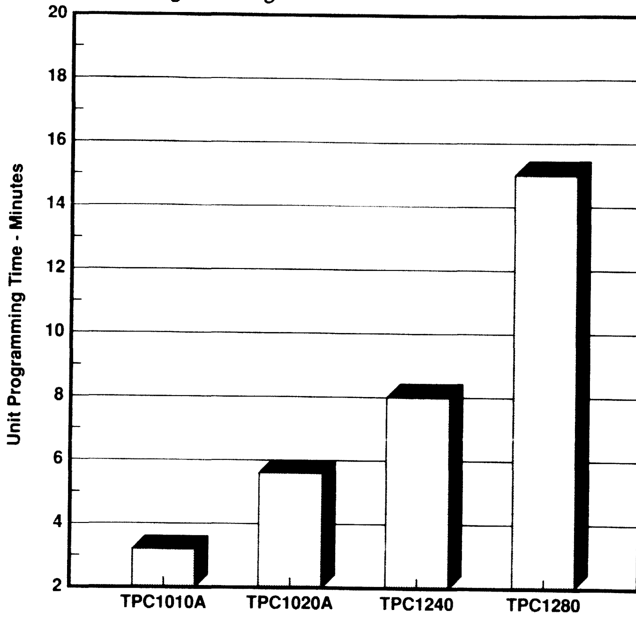
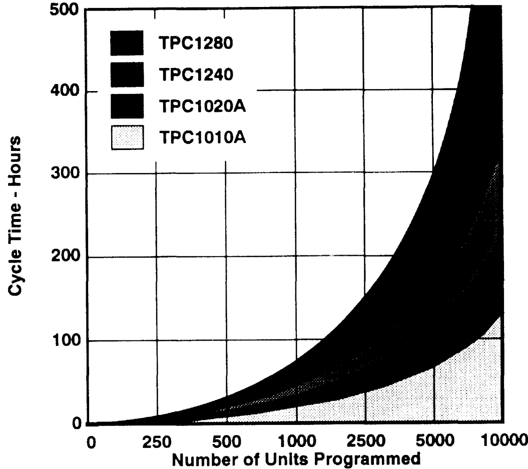


Figure 7-2 shows programming cycle time as a function of the quantity of units programmed. The data shown is based on the use of one TI Activator 2 (gang 4) programming unit.

Figure 7-2. FPGA Programming Cycle Time



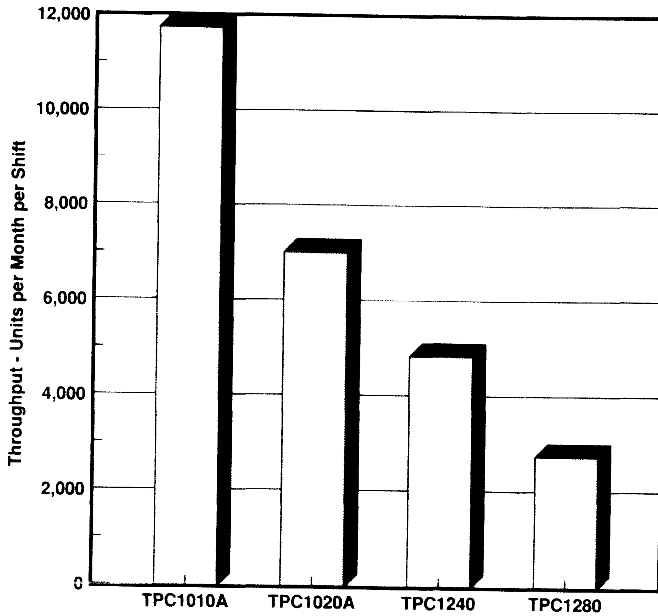
As indicated in Figure 7-2, programming cycle time becomes significant as run rates approach 500 units. The following total programming time is required to complete 500 units:

- 6.9 hours (TPC1010A)
- 11.5 hours (TPC1020A)
- 16.7 hours (TPC1240)
- 31.3 hours (TPC1280)

Programming time thus can have a critical effect on production scheduling and capacity. Because programming throughput degrades as programming time increases, carefully analyze the anticipated volume and device mix to determine the amount of capacity required for an in-house programming operation.

Figure 7-3 shows the throughput for a one-shift operation using a single Activator 2 programming unit based on a 7.3-hour shift and 22-day month.

Figure 7-3. Activator 2 Programmer Throughput

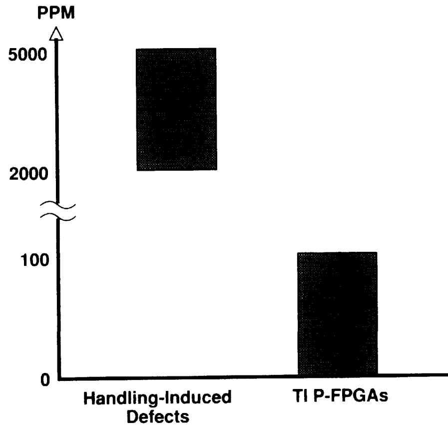


7.1.2 Surface-Mount Component Lead Quality

Maintaining the lead quality of surface-mount components is another key factor in implementing an in-house programming operation. Degradation in lead quality can result from handling surface-mount components. The number of handling-induced defects depends on the number of process steps and whether handling is manual or automated.

Figure 7-4 shows the effect of device handling on lead quality, based on TI's internal 68- and 84-pin PLCC manufacturing operation.

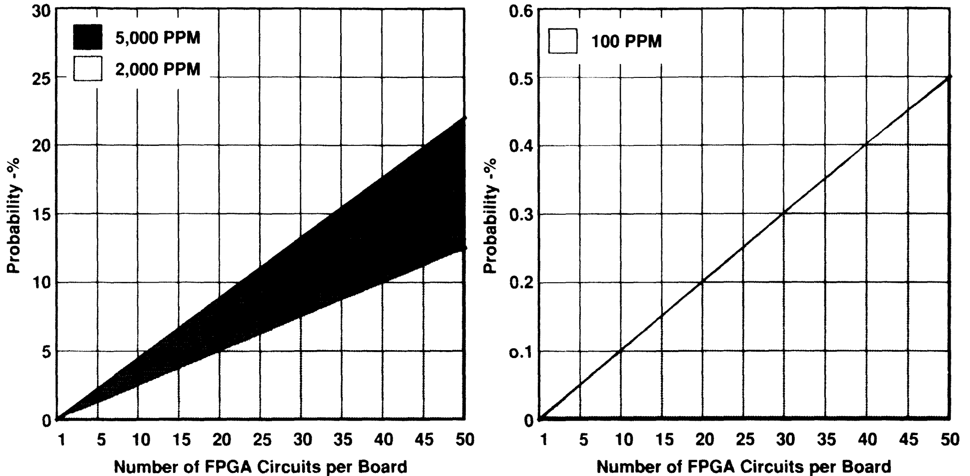
Figure 7-4. 68-/84-Pin PLCC Defect-Level Leads



TI P-FPGA devices undergo extensive lead conditioning and inspection prior to shipment, resulting in significantly lower defect levels.

The significance of defect level on the board assembly process is shown in Figure 7-5, which indicates the probability of finding at least one defective out-of-spec device on a single board. This probability is indicated as a percentage and is a function of defect level and number of units per board.

Figure 7-5. Defect Probability and PPM Level



The probability of finding one defective unit on a board with ten FPGAs is:

- 5 percent (1 in 20) at 5,000 ppm
- 2 percent (1 in 50) at 2,000 ppm
- 0.1 percent (1 in 1000) at 100 ppm

In contrast to the high probability of board assembly problems related to increased lead defects, TI P-FPGAs virtually eliminate lead defects. As a result, your design will achieve lower cost of ownership along with increased manufacturing efficiency.

7.1.3 Handling Moisture-Sensitive Plastic-Surface-Mount Components

Another important manufacturing consideration is the required handling procedure for moisture-sensitive plastic-surface-mount components (PSMCs). Moisture absorbed from the atmosphere by plastic-encapsulating integrated circuits vaporizes during the solder reflow process and can cause the package to crack. Special handling and packing procedures must be followed to prevent PSMC moisture absorption.

All TI plastic surface-mountable FPGAs and P-FPGAs are baked until the moisture content of the plastic package is reduced to under 0.05 percent and shipped in heat-sealed dry-pack bags. Humidity indicator cards are sealed with the components to warn of seal failure and exposure to moisture.

Moisture absorption, the amount and rate of which is a function of temperature and relative humidity, begins immediately upon exposure of the PSMC to the environment. The maximum allowable exposure time after the bag is opened at 30°C and 60 percent relative humidity is 48 hours.

Because an in-house programming operation necessarily exposes FPGAs to the open air, the exposure time limit is an important consideration. Production programming capacity, scheduling, and cycle time must be appropriately planned to prevent the addition of a rebake process step. Bake and dry-pack equipment are needed in case maximum exposure time is exceeded or programmed devices are inventoried for later use.

7.1.4 Manufacturing Efficiency

The complexity of FPGA in-house programming increases with quality and manufacturing requirements. In addition, the symbolization of programmed devices is necessary to prevent mixing of different designs.

If pick-and-place equipment is used for PLCC packages, programmed FPGAs may have to be taped and reeled prior to use. In this case, material has to be routed to in-house tape and reel equipment or to an outside subcontractor. Regardless of the method you choose, increases in cost and process complexity are the results.

The following list sums up in-house programming considerations:

- Processing requirements and capacity
- Programming and symbolization equipment
- Handling procedures to minimize lead degradation, rebake, EOS, and ESD damage
- Outgoing quality
- Staffing requirements
- NRE, capital, overhead, and labor costs

Your decision to purchase TI factory programmed FPGAs requires careful consideration of the following factors:

- Cost adders
- Volume requirements
- Outgoing quality
- Leadtime

Moving programming, symbolization, and tape and reel back to TI will substantially reduce your design's process steps, material handling, and defects. P-FPGAs are received and placed in inventory until needed for production.

Figure 7-6 illustrates the reduction of in-house processing achieved by using TI P-FPGAs. The left column shows the steps typically required to program units in-house. The right column shows the steps required if TI programs the units for you.

Figure 7-6. Blank vs. TI-Programmed FPGA Process Steps

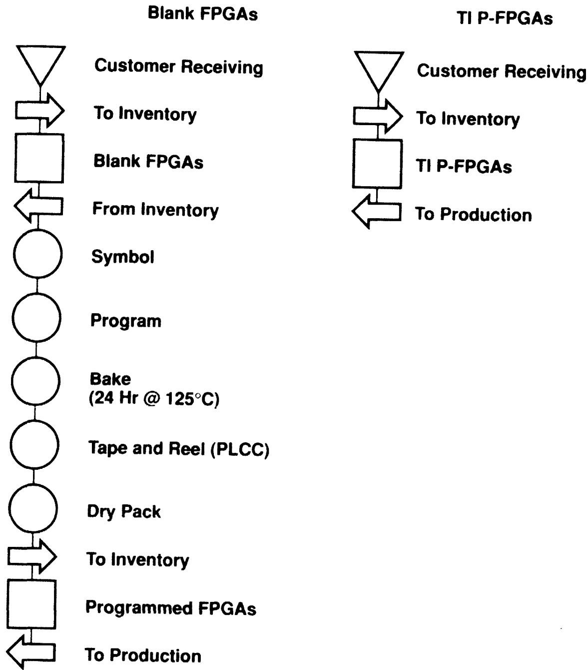
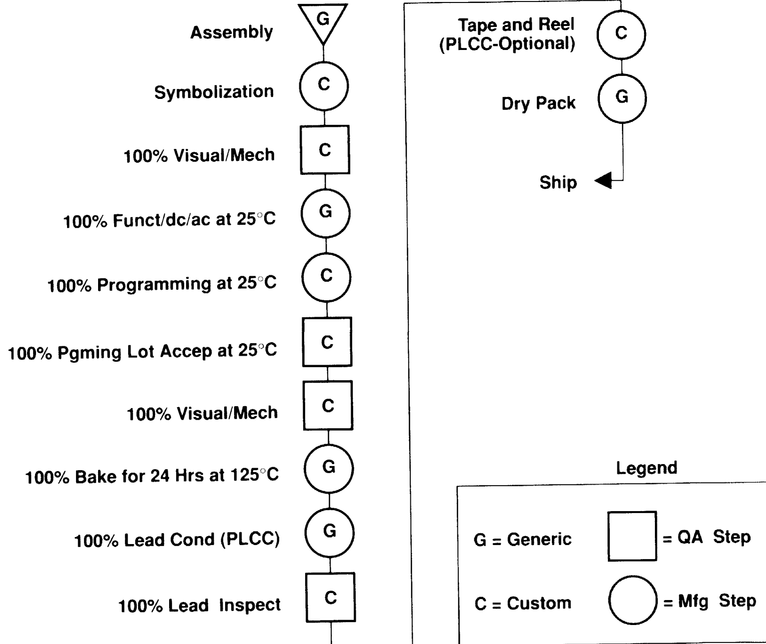


Figure 7-7 shows the TI P-FPGA manufacturing flow implemented to provide the highest quality product possible.

Figure 7-7. TI P-FPGA Factory Process Flow



7.2 How TI Tests FPGAs†

The TPC10 and TPC12 series families of FPGA devices are designed with a comprehensive and thorough set of testing circuitry providing 100 percent fault coverage of all logic prior to programming, thus eliminating the requirement to functionally test the device after programming.

This section describes the various FPGA design-for-testability techniques and how they are used during factory testing. For information on temperature testing, parametric specifications, etc., refer to the datasheet.

7.2.1 Serial-Test Circuitry

Internal test observability and controllability of TI FPGA devices is performed via a long serial shift register chain surrounding the array, the length and partitioning of which are determined by the number of gates and the device type.

The only package pin that you cannot program (other than V_{CC} and GND) is the *MODE* pin, which is dedicated to the selection of test, programming, or debug operation at high level (V_{CC}) and normal operation at low level (GND). The *MODE* pin must not be allowed to float and must be tied to GND during normal operation.

Test data is serially shifted into the *SDI* pin one bit at a time by clocking the *DCLK* pin. Subsequent test results are serially shifted out at either the *SDO* pin or the *MPR* pin also by clocking the *DCLK* pin. Only the I/O input and output buffer tests are functionally tested at each individual I/O pin.

This shift register is divided into several different groups for access to the top, bottom, left, right, middle, and center of the array. The first block of registers is the mode control, which, once loaded, determines the setup and sequence of events to occur during the test. The other blocks provide each channel and column track with the ability to be forced or sensed in any combination or direction with V_{PP}, H, L, Z, or a precharged condition. The new data can then be loaded back into the shift registers, shifted out, and read.

† Contributed by Jim Ptasinsky, Engineering, Texas Instruments Incorporated.

7.2.2 Test Flow

The factory test flow used on TI FPGA devices in the unprogrammed state is described in this section by the following major test categories:

1) Pin-to-pin opens and shorts test
2) Static power-pin parametric current test
3) Serial shift register chain functionality test
4) Channel and column tracks opens, shorts, and leakage tests
5) Array transistors functionality and leakage tests
6) Logic modules addressing, functionality, and microprobing tests
7) High-voltage junction and antifuse stress tests
8) Antifuse addressing and shorts test
9) Silicon-signature and bin-circuit programming tests
10) I/O buffers functionality/parametrics/3-state leakage tests

7.2.2.1 Device Pin-To-Pin Opens and Shorts Test

The pin-to-pin opens and shorts test, the most basic test performed on an integrated circuit, checks for assembly and handling defects and ESD/EOS damage near the internal bonding pads of the device.

The most common problems associated with FPGA programming are caused by mechanical damage to the package pins or leads. The high pin count and fine pitch used on flatpacks and chip-carrier packages require close attention during socket insertion on the Activator programming unit.

Other defects found by this test are caused by electrostatic discharge into a device pin, caused by improper handling or electrical overstress.

7.2.2.2 Static Power-Pin Parametric Current Test

Each power pin is parametrically measured for the amount of static and quiescent current used to meet the specified maximum value.

7.2.2.3 Serial Shift Register Chain Functionality Test

The serial shift register chain functionality test must be performed before other functional tests can be validated. No other addressing or readback will be reliable or even possible until the shift register has passed every test and is100-percent functional.

7.2.2.4 Channel and Column Tracks Opens, Shorts, and Leakage Tests

The TI FPGA architecture includes horizontal and vertical metallization occupying a significant portion of the die area and is a factor in your design, allowing the devices to achieve 80-95 percent module utilization. All interconnection and routing tracks must first be completely tested for connectivity, opens, and shorts .

Precharging the lines checks for leakage paths and device quality. The test charges up each track and maintains the level for a predetermined time, during which the charge must remain high enough to allow a pass. The discharge rate is directly related to the integrity of the chip and provides a quality indication of many of the key process parameters.

7.2.2.5 Array Transistors Functionality and Leakage Tests

The horizontal tracks are divided optimally with segmentation transistors. Each of these horizontal pass transisters are in parallel with an antifuse called the *H fuse*. In addition, the vertical tracks have many vertical pass transistors in series along with a *V fuse* in parallel. The gates of these transistors are turned on and off in accordance with the register sequence and the test mode. All array transistors are therefore fully tested for correct switching.

Track testing and array testing are not entirely independent. The test flow builds upon itself by using previously tested and known working paths and devices to check the next unknown level. Test vectors are designed to isolate and differentiate between many defects, such as an open track or open transistor; this also applies to shorts and leakages in a track transistor.

7.2.2.6 Logic Modules Addressing, Functionality, and Microprobing Tests

After the entire routing, control, and test infrastructure of the die is determined to be functional and free of defects and leakage, you can test the full functionality of each logic module in the array.

The eight inputs and one output of each module are toggled through all combinational or sequential (if applicable) truth tables and states via the serial shift register.

The microprobing and diagnostic capability of each module is also checked to ensure that any two module outputs throughout the array can be diagnosed by you after the device is programmed.

7.2.2.7 High-Voltage Junction and Antifuse Stress Tests

Stress testing ensures the highest possible antifuse reliability and high-programming yield. High-level voltages are used in the 12-V to 21-V range on the V_{PP} pin. Each transistor involved in device programming is subjected to junction stress for a predetermined time as a check for low-voltage breakdown conditions.

Antifuse stress testing weeds out process defects that can cause a weak fuse. Half the V_{PP} voltage level required to program a fuse is used, which is important because addressing and selecting a single fuse during programming subjects other fuses on the intersecting tracks to a voltage level of $V_{PP}/2$ for a period of time during programming.

7.2.2.8 Antifuse Addressing and Shorts Test

Antifuse shorts testing is commonly known as the blank test because of its similarity to the blank test performed on the device by the Activator programming unit.

The previous antifuse stress tests apply stressing levels for predetermined times without shifting-out data. The check for a resulting short is tested during the blank test.

7.2.2.9 Silicon-Signature and Binning-Circuit Programming Tests

The few differences between the test flow performed on each die during wafer sort and final test on each package occur during these tests. In the wafer sort flow, some words of the silicon signature may be programmed with factory-specific codes and information that provide die traceability to the process fab, lot, and wafer. In addition, a code is programmed to establish the V_{PP} level to be used during programming. This information can be re-read when necessary on the Activator programming unit or at final test.

The binning circuit consists of several extra testing modules configured during sort to produce a simple series of gates between the *bin-in* and *bin-out* pins. At final test, ac propagation delay measurements are made to bin each packaged device into various speed-grade categories. Each unit is then symbolized with an add-on dash number representing the overall performance range of each device.

7.2.2.10 I/O Buffers Functionality, Parametrics, and 3-state Leakage Tests

The last group of tests checks the functionality of each input, output, and clock network buffer. The parametric V_{OH}/V_{OL} and 3-state leakage tests performed at this time are the only tests in which the remainder of the device pins are tested and actually used.

The functional tests described in the previous sections are performed using the *MODE*, *DCLK*, *SDI*, *MPRA*, and *MPRB* pins. The TPC12 series family also has an *SDO* pin.

The fully integrated test, programming, and operational architecture and circuit techniques of TI FPGA products offer the highest possible quality and reliability with preferred one-time-programmable antifuse technology.

7.3 Programming and Verifying FPGA Antifuses†

TI FPGA devices utilize antifuse technology rather than the fusible-link structure common to most PLDs. This fundamental difference between normally open antifuses and normally closed fuses provides FPGAs with near-flawless design-for-testability circuitry. The inherent nature of antifuse-based FPGA architectures results in complete postprogramming functional capability while maintaining the preferred one-time programmable feature.

This section focuses on fuse verification testing performed during the Activator programming process.

7.3.1 Activator Programming Procedure Overview

The first step performed by the Activator hardware programming cycle is to read the silicon signature of each device in each socket to ensure that the device is inserted properly and that the device type matches the project.

Note:

If the security fuse option is used and successfully programmed, any attempt to read the silicon signature or checksum will be disabled, and the display will erroneously indicate that the device is incorrectly inserted.

The second step before programming is to determine whether the device is blank; i.e., not previously programmed. The blank test is executed using a highly parallel technique for the fastest response time. If one or more fuses are found to be programmed, the Activator programming unit will display the message `NON-BLANK` and abort.

The blank test does not determine which fuses are shorted. In addition, there are no provisions for reading or displaying a fuse map as on many PLD device programmers; thus, because the place-and-route algorithm creates a unique order in the fusing sequence for every design, an unknown design and programmed FPGA device cannot be verified or have its fuse map extracted. Without knowing the `.fus` file used, the system cannot determine the verification paths to address each programmed fuse.

After the device passes the initial blank test, the programming cycle starts (see Section 7.3.2). The system will stop at the first indication of a failed test, and an error display will appear. You are encouraged to label failures where the failure occurred with the design name and the fuse and test numbers.

† Contributed by Jim Ptasinsky, Engineering, Texas Instruments Incorporated.

7.3.2 Antifuse Programming

Programming begins when the system puts the device into the programming mode, loads a fuse address starting with the first fuse in the `.fus` design file, and proceeds until all fuses and tests pass successfully. A single fuse at a time is addressed and programmed; the rest are biased at safe levels. The fuse can program with as few as one high-voltage V_{PP} pulse or as many as hundreds, up to a maximum allowable value.

If the maximum allowable value is reached before the fuse is programmed, the error message `prog-fuse-fail` will be displayed along with the failed fuse number. The fuse number is the numerically consecutive number in the unique design `.fus` file and varies from code to code; thus, a particular fuse number should never be the same fuse location on the die.

After the fuse address is applied, a lower-level V_{PP} is applied to measure I_{PP} current through the selected fuse. The value read should be near zero to indicate an initial open, or unblown, antifuse. This method is used for both preprogrammed and programmed fuse verification. If an initial current is read before the first programming pulse occurs, the error message `integrity check 6` will be displayed for that device.

7.3.3 Antifuse Verification During Programming

TPC10 series devices contain several little-known tests that take place before, during, and after each fuse is programmed to ensure the highest possible postfunctional success and performance for each device. Antifuses are verified electronically, either by logical sensing or current sensing.

Current sensing, which is used with TI FPGA, is the most versatile method, considering the amount of information obtained and control of the power each fuse is subjected to. Most PLD chip designers use a binary high-and-low functional output to determine whether a fuse is programmed.

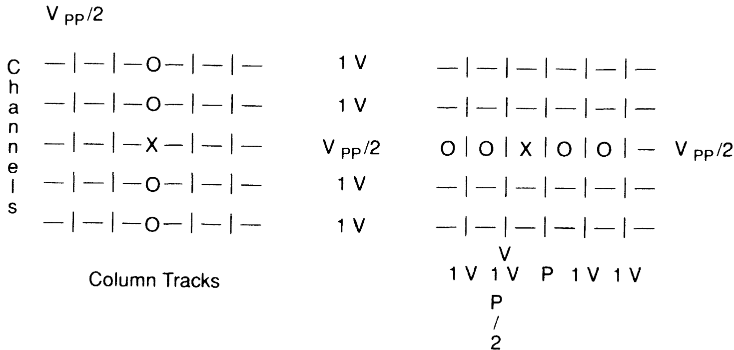
TI FPGAs allow continuous µA current sensing of the V_{PP} pin in the Activator programming unit during programming, giving the system the ability to utilize many different thresholds based on the analog nature of current sensing.

Inherent variations in the programming paths of each fuse in the array can be overcome by optimizing the applied voltage and current levels for the various fuse types and locations throughout the device. These techniques achieve the goal of controlled fuse resistance at the lowest attainable levels and produce the highest and most predictable performance and timing.

7.3.4 Antifuse Integrity Test 7 and Integrity Test 8

Integrity test 7 and *integrity test 8* are the two most important tests performed after each antifuse is programmed (Figure 7-8).

Figure 7-8. Antifuse Integrity Test 7 and Integrity Test 8



Before the next fuse to be programmed is addressed, the system will check the integrity of the unblown fuses along the same channel and column track.

Integrity test 7 rechecks to see if the fuses above and below the target fuse on the same column are still open. All of these fuses are addressed together to measure current flow through V_{PP}. If current flow is detected, programming will stop, the system will abort, and the integrity test will be displayed along with the last fuse number programmed.

Although similar to *integrity test 7*, *Integrity test 8* addresses unblown fuses along the horizontal channel track to the right and left of the fuse just programmed.

7.3.5 Browsing Your .AVI File

During your programming cycle on the Activator programming unit, a file is updated and stored in your `design_name` directory with the extension `.avi`. This file records valuable programming and verification data that has just been generated. Use the **Browse** command or exit to DOS to read the file, especially if a failure occurs.

On the Activator 2 programming unit using the APS 2.1 version 3 software revision or later, the `.avi` file is organized into eight columns of data following each of the fuse numbers. As you read from left to right, each of the four pairs of data corresponds in reverse order to the four socket numbers; i.e., socket #4, #3, #2, and #1.

The first number of each pair is the final V_{PP} current through the fuse. The second number of each pair is the total V_{PP} pulses applied to the fuse in that socket. For example, 112 33 means that 11.2 mA programmed the fuse after 33 pulses.

Note:

Remember to place the decimal point one digit to the left to read the result in milliamperes.

If a *programming fuse fail* display occurs at the last fuse number of a certain socket, the `.avi` file will show 0 20000 as a possible result. If an integrity test fails, the fuse data will usually appear normal, which means that one of these tests detected current elsewhere, as previously explained.

The activate cycle can be restarted from the last fuse of a certain socket by entering

```
act-fuse fuse# 0 1
```

Start at the same fuse number when that fuse did not program at all. This will cause the hardware to apply more fuses to that fuse. For integrity test fails, start at the next fuse in the sequence.

7.4 Speed-Enhanced FPGA Devices†

TI FPGA devices are sorted by their relative speed/performance. The most common speed bins are:

1) Standard device
2) Dash-1 (15% higher-speed-grade device)
3) Dash-2 (25% higher-speed-grade device)

A standard device has no dash number following its device symbolization; a *-1* or *-2* follows the device type of the higher speed grade devices.

7.4.1 Why Speed Binning

Designs that consolidate most or all of the logic into FPGAs will occasionally need speed-sorted devices; that is, a particularly critical path or location on the board will need to be faster than the rest.

7.4.2 TPC1010A and TPC1020A Binning Circuits

Speed-binning of TI-FPGA devices is provided through by a factory-programmed binning path on each die at wafer sort. This binning circuit consists of 16 modules on the TPC1010A, with a column of test modules on two sides of the array. The TPC1020A has 28 modules configured similarly. In addition, in each path there is an associated delay of one input buffer and one output buffer. All bin modules are programmed into alternating inverting and noninverting buffers, each with a fanout of one.

After the binning circuit is programmed, the testing function provides an ac propagation delay measurement path from the bin input pin, *BININ*, to the bin output pin, *MPRA*. The speed is then measured through the binning circuit during a package-level final test on each unit. Afterward, the sorted units are symbolized accordingly with the appropriate dash number. See Table 7-1 for binning path pin assignments.

Note:

Testing can also be performed after you have programmed the array.

† Contributed by Jim Ptasinsky, Engineering, Texas Instruments Incorporated.

Table 7-1. TPC10 Series Binning Path Pin Assignments

Pin Name	44 Pin	68 Pin PLCC	84 Pin	84 Pin CPGA	100 Pin PQFP
MODE	34	54	66	E11	92
SDI	36	56	72	B11	98
DCLK	37	57	73	C10	99
BININ 1010A	9	12	NA	D2	34
BININ 1020A	7	10	13	C2	31
MPRA/BINOUT	38	58	74	A11	100

7.4.3 Setup and Measurement Procedure for the TPC10 Series

Perform the following setup and measurement steps.

Step 1: Follow the normal device power-up procedure in a suitably designed test fixture. Connect all of the GND pins to the ground potential and set the *MODE*, *DLCK*, and *SDI* pins initially to a V_{IL} between 0 V and 0.3 V. Raise all of the V_{CC} pins and V_{PP} to 5.0 V.

Step 2: Raise the *MODE* pin from its initial value (0 V to 3 V) to put the device in the test mode.

Step 3: Load the 7-bit mode register block of the serial-shift register chain with the following sequence: 1011101. To do this, apply a bit, then clock, and follow with the next bit until seven clocks are performed. Use a clock rate of less than 1 MHz and setup and hold times of approximately 0.2 μs, which puts the device into the binning measurement state.

Step 4: Apply a pulse to the *BININ* pin and observe the output. noninverting propagation delay can now be measured.

Note:

The t_{plh} and t_{phl} readings are generally not equal. The factory uses the slowest of the two reads to sort each device.

7.4.4 TPC12 Series Speed Binning

Because of the complex setup and measurement required for the TPC12 series speed binning, TI does not recommend that you perform this procedure.

7.5 Automatic Test Pattern Generation (ATPG)†

This section describes the design flow used to migrate an FPGA design to a TGC1000 ASIC using the Synopsys design compiler and test compiler. Scan circuitry is automatically inserted into the translated netlist. ATPG produces TDL'91 format test vectors for 100 percent fault coverage of the design.

7.5.1 Testability

Because the internal nodes of an ASIC device can only be accessed using expensive equipment, economy dictates that an ASIC design circuit be capable of being debugged without physically contacting the intermediate nodes.

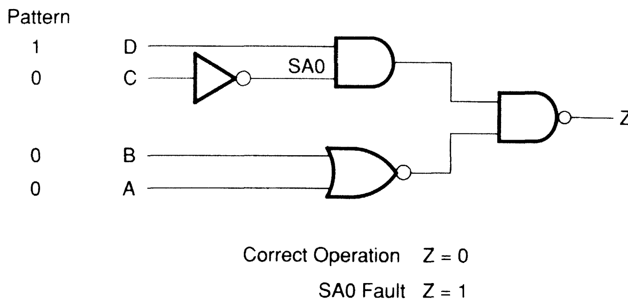
You must consider testability early in the design. Poor testability can mean poor manufacturing yield and an inability to manufacture the device in volume. In addition, the time required for debugging should be minimized in the early stages of the design.

7.5.2 Fault Model

The goal of testing an ASIC is to expose physical defects on the silicon after manufacture. The *stuck-at-fault* model is a familiar approach to this problem. A node is *stuck at* 0 if it remains at a logical 0 when controlling signals should cause it to switch in normal operation. Test patterns are developed to detect as many of these faults as possible.

Figure 7-9 shows how a fault stuck at 0 on a node can be detected using the test pattern shown. The effect of the fault is propagated to the output Z.

Figure 7-9. Stuck-at-Fault Model



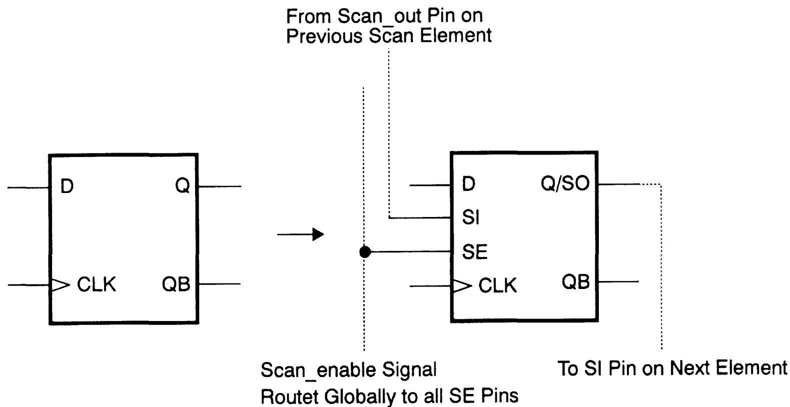
† Contributed by Katey Derbyshire, FPGA Applications, Texas Instruments Ltd.

7.5.3 Scan Design

Design-for-test ensures that controllability and observability are maintained and that the value on each internal node can be set by the primary inputs and observed by the primary outputs. Scan design, in which sequential elements in a circuit are replaced with scannable equivalents, is one of the most widely used design techniques used to achieve this criteria.

Several scan implementation alternatives; e.g., multiplexed flip-flop, clocked scan, and LSSD, are supported by the Synopsys test compiler. Figure 7-10 shows how a D-type flip-flop is modified for scan. The *scan_enable* signal selects data from the *d* input during normal operation and from the *scan_in* input during test mode. The input data during test mode comes from the *scan-out* pin of the previous element in the chain, unless it is the first element, in which case it will be a primary input.

Figure 7-10. Multiplexed Flip-Flop Scan Methodology



Scan elements are connected together into one or more scan chains. Each sequential element can be set to a known state by serially shifting in the appropriate values via the primary inputs.

Internal nodes can be observed at the primary outputs by latching the outputs of combinational logic blocks into the scan elements and serially shifting them out through the scan chain.

Successful scan design relies on the ability to control and observe the internal states of a design. Design rules include a restriction against gated clocks and combinational feedback loops. Prior to scan insertion, the Synopsys test compiler checks for design rule violations that can affect testability.

7.5.4 Automatic Test Pattern Generation

Test patterns are developed by scan design based on two assumptions: all inputs to combinational logic are controllable, and all outputs are observable. Each test pattern consists of a set of values applied to the primary inputs controlling a particular node of the design and to the corresponding expected outputs.

Each node is checked for both a stuck-at-0 fault and a stuck-at-1 fault. During device testing, measured outputs are compared to expected outputs; if they do not match; a fault will be recorded. Test patterns that can detect all the possible stuck-at-0 and stuck-at -1 faults in a design provide100-percent fault coverage.

The Synopsys test compiler generates test patterns for both stuck-at-0 and stuck-at-1 faults on each cell pin in three stages. In the first stage, the test compiler generates patterns at random. Fault coverage is monitored continually and when it is no longer increasing, the test compiler moves onto the second stage.

In the second stage, the test compiler looks at each undetected fault individually and tries to generate a pattern to cover it. If a pattern cannot be generated for a particular fault, the fault is considered to be undetectable and reported as such.

In the third and final stage, the test compiler reduces the generated patterns to the minimum number required to obtain the maximum fault coverage.

In addition to generating patterns that test the existing logic in the design, the test compiler also generates patterns that test the scan circuitry itself. These two pattern sets are combined and converted into the required tester format. In this application, the TI TDL'91 format is used.

7.5.5 Migration Flow

Figure 7-11 shows the design flow used to migrate the application example from an FPGA to an ASIC. The original FPGA netlist is retargeted to the TGC1000 technology using the Synopsys design compiler. Figure 7-12 shows a script that automates this translation. The target library is now TGC1000 and constraints are applied to the translation to reflect realistic operating conditions. A new TGC1000 Verilog netlist is written.

Now the scan-test methodology for the design is selected by entering the command (a multiplexed flip-flop is used in this example):

```
set_test_methodology multiplexed_flip_flop
```

The design is checked against the design rules of the particular scan-test methodology chosen. The **check_test** command applies the appropriate set of rules in sequence to the design. Any errors or warnings resulting from design rule violations are reported. Errors must be corrected before continuing to the next stage.

Figure 7-11. FPGA-to-ASIC Design Flow

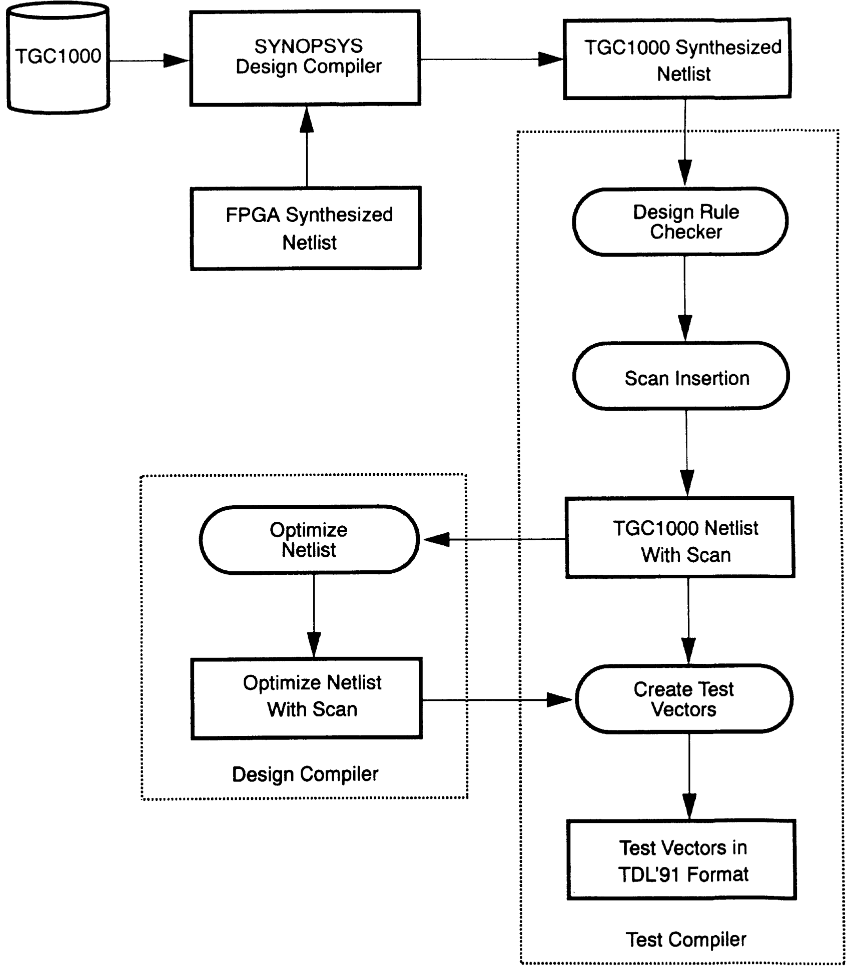


Figure 7-12. Synopsys Script for TPC10 to TGC1000 Translation

```
/* Set up Translation Libraries and Read FPGA Verilog Netlist */
remove_design
link_library = {GCELL.db TPC2TGCHM.db TPC10.db TGC1000_COM_MAX.db}
target_library = {TGC1000_COM_MAX.db}
read -format verilog TIMER_COMP.v

/* Set Constraints for Translation to TGC1000 */
current_design = TIMER_COMP
check_design > check_tgc.rpt
set_max_fanout 6.0 "CORE"
set_drive drive_of(TGC1000_COM_MAX.db/AN210/Y) all_inputs()
set_load load_of(TGC1000_COM_MAX.db/BU130/A) all_outputs()

/* Translate, report, and write TGC1000 Verilog */
translate -verify
current_design = TIMER_COMP
report -area -constraint -timing > check_tgc.rpt
write -h -f verilog -o TIMER_COMP_TGC.v TIMER_COMP
```

The *check_test* output can be redirected to a file by entering:

check_test > testrules.rpt

Scan-test circuitry is then automatically inserted into the design in the scan implementation style chosen. The **compile_test** command is invoked at the level below the I/O collar; in this way, scan will be inserted into the core logic and the I/Os will not be affected.

The target library containing the scan cells must be specified prior to compilation. When the **compile_test** command is executed, sequential cells in the design are replaced by their scannable equivalents; test control and scan signals are routed to each of these cells. Adding scan elements changes the area and performance of a design. Reoptimization using the Synopsys design compiler may be required to meet existing system design constraints.

Figure 7-13 shows how the commands described above have been applied to this application example and suggests other commands that can be used.

Figure 7-13. Script for Scan Insertion Using the Synopsys Test Compiler

```
/* Set up Libraries */
remove_design
link_library = { TGC1000_COM_MAX.db TGC1000_TC.db }
target_library = { TGC1000_COM_MAX.db TGC1000_TC.db }
symbol_library = { TGC1000.db }

/* Set up Test Compiler Parameters */
test_scan_in_port_naming_style = "TEST_SI_%s%s";
test_scan_out_port_naming_style = "TEST_SO_%s%s";
test_scan_enable_port_naming_style = "TEST_SE_%s%s";

/* Put Scan Chains into the Core */
read -format verilog TIMER_COMP_TGC.v
current_design = CORE
set_test_methodology multiplexed_flip_flop
check_test -verbose > testrules.rpt
compile_test -no_disable
report_test -scan_path
current_design = TIMER_COMP
link -all
write -h -f verilog -o TIMER_SCAN .v
```

To illustrate the test compilation process, consider the state machine block of the 12-hour clock. Figure 7-14 shows the block synthesized to the FPGA TPC10 series technology. Note the Boolean functions, such as the NAND3A and NOR3B that are specific to the FPGA library, and the DF1 D-type flip-flops.

Figure 7-14. TPC10 Series Timer State Machine Schematic

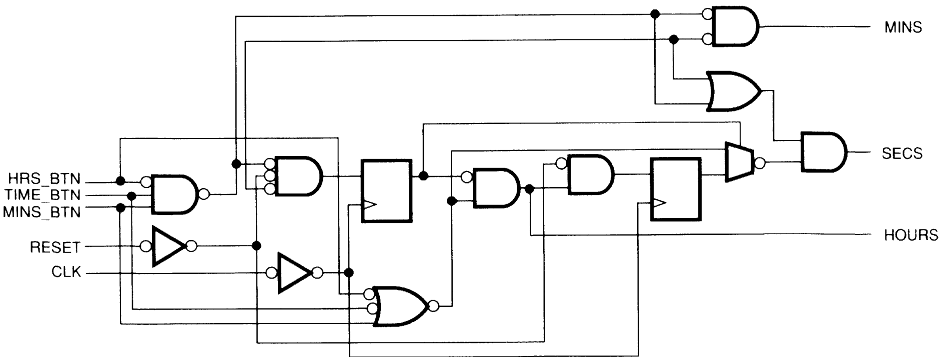
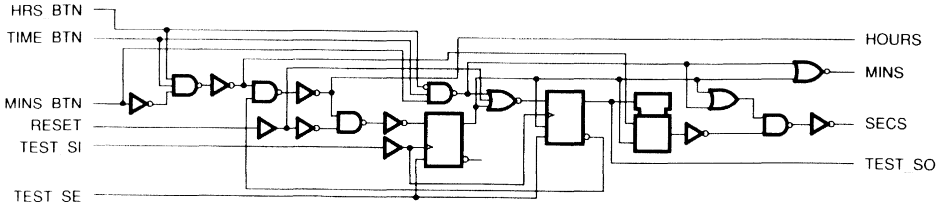


Figure 7-15 shows the same block after translation to TGC1000 and scan insertion by test compiler. The two DF1 flip-flops have been translated to SDTN00 scannable equivalents and the *TEST_SI*, *TEST_SE*, and *TEST_SO* test mode signals added.

Figure 7-15. TGC1000 Timer State Machine Schematic With Scan



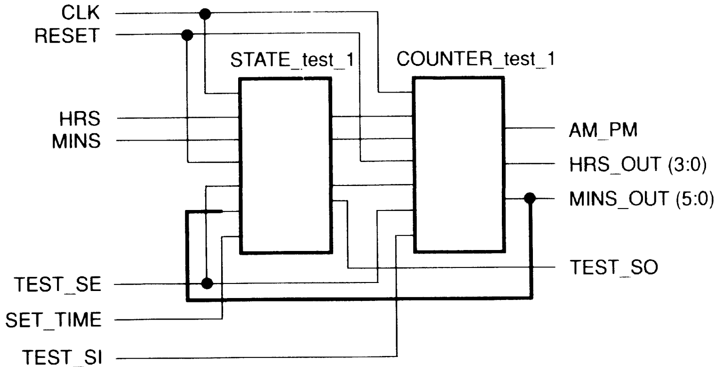
The **`report_test -scan_path`** command generates the scan path report, which lists the *scan_in* and *scan_out* pins of every element in the scan chain as well as the *scan_in* and *scan_out* ports (Figure 7-16). The scan path report lets you verify that all sequential cells in the design are made scannable and included in the scan chain.

Figure 7-16. Scan Path Report Generated by the Test Compiler

```
****************************************
Report  : test
-scan_path
Design  : CORE
Version: v2.2a-6762
Date    : Wed Oct  7 21:44:48 1992
****************************************
Scan path contains 19 cells:
TEST_SI         ->
COUNT1/AM_PM_reg/SD - Q          ->
COUNT1/CR_SECS_reg[0]/SD - Q  ->
COUNT1/CR_SECS_reg[1]/SD - Q  ->
COUNT1/CR_SECS_reg[2]/SD - Q  ->
COUNT1/CR_SECS_reg[3]/SD - Q  ->
COUNT1/CR_SECS_reg[4]/SD - Q  ->
COUNT1/CR_SECS_reg[5]/SD - Q  ->
COUNT1/HRS_OUT_reg[0]/SD - Q  ->
COUNT1/HRS_OUT_reg[1]/SD - Q  ->
COUNT1/HRS_OUT_reg[2]/SD - Q  ->
COUNT1/HRS_OUT_reg[3]/SD - Q  ->
COUNT1/MINS_OUT_reg[0]/SD - Q  ->
COUNT1/MINS_OUT_reg[1]/SD - Q  ->
COUNT1/MINS_OUT_reg[2]/SD - Q  ->
COUNT1/MINS_OUT_reg[3]/SD - Q  ->
COUNT1/MINS_OUT_reg[4]/SD - Q  ->
COUNT1/MINS_OUT_reg[5]/SD - Q  ->
STATE1/CR_STATE_reg[0]/SD - Q  ->
STATE1/CR_STATE_reg[1]/SD - Q  ->
TEST_SO
```

Figure 7-17 shows the level of hierarchy above the state machine. The scan path can be traced from the *TEST_SI* input, through the *COUNTER* block via the *MINS_OUT[5]* signal, into the *STATE* block, and finally out through the *TEST_SO* output. At this stage it is necessary to manually add input and output buffers at the top level for the additional test ports created during scan insertion. This step is not required if the test compiler is instructed to use existing pins.

Figure 7-17. Timer Core Block Showing the Scan Path



The final step of the migration flow is to generate manufacturing test vectors in two parts:

1) Generate a set of test patterns with high fault coverage.
2) Format the vectors into TI's TDL'91 format.

The test compiler automatically generates test patterns that target all of the stuck-at faults in a scan-based design. The test compiler generates patterns that test both the scan circuitry and the combinational logic of the design.

The scan circuitry is tested by shifting in an alternate pattern of 1s and 0s through the scan path. A fault will be recorded if the sequence is incorrect at the *scan_out* port.

The combinational logic is tested by applying the generated set of test patterns. Each pattern is applied in the following sequence:

1) The test pattern is serially shifted into the scan chain through the *scan_in* port.
2) Stimulus is applied to the primary inputs.
3) The output response is measured.
4) The system clock is pulsed once to latch all the internal states.
5) The response is shifted out to the *scan_out* port.

These five steps constitute one tester cycle and are repeated for each test vector.

The aim of ATPG is to generate a minimum set of test vectors for all possible stuck-at faults in the design. Figure 7-18 shows the final script used in the design flow.

Figure 7-18. Synopsys Script to Generate TDL'91 Test Vectors

```
/* Set up Libraries */
remove_design
link_library = { TGC1000_COM_MAX.db TGC1000_TC.db }
target_library = { TGC1000_COM_MAX.db TGC1000_TC.db }
symbol_library = { TGC1000.db }

/* Enable TDL'91 Test Vector Format */
write_test_enable_tdl91 = TRUE;
write_test_default_strobe = 1000.0;
write_test_default_strobe = 900.0;

read -f verilog TIMER_SCAN.v
current_design = TIMER_COMP
set_test_methodology multiplexed_flip_flop -existing_scan
set_signal_type test_scan_in TEST_SI_I
set_signal_type test_scan_enable TEST_SE_I
set_signal_type test_scan_out TEST_SO_I
check_test -verbose > chktstTIMER.rpt
create_test_vectors
report_test -scan_path
report_test -coverage -faults -class untestable > test.rpt
write_test -format tdl91
```

The test compiler command, **`create_test_vectors`**, generates the test vectors for the design using a combination of random and deterministic pattern generation, provides the status of individual faults in the circuit, and reports the overall fault coverage achieved at each stage.

The **`report_test`** command lets you examine the fault coverage in more detail.

Figure 7-19 shows the report obtained from the script in Figure 7-18.

Figure 7-19. Test Coverage Report

```
****************************************
Report : test
-coverage
Design : TIMER_COMP
Version: v2.2a-6762
Date    : Tue Sep 22 11:16:59 1992
****************************************

    Detect    : Number of faults Detected
    Aband     : Number of faults Abandoned
    Tied      : Number of faults Tied High or Low
    Redund    : Number of faults Redundant
    Untest    : Number of faults Untested
    Total     : Number of faults Total
    Coverage : Detect / (total - Tied - Redund)
```

Design/Cell	Detect	Aband	Tied	Redund	Untest	Total	Coverage
IN_HRS	4	0	0	0	0	4	100.00%
IN_MINS	4	0	0	0	0	4	100.00%
IN_RESET	4	0	0	0	0	4	100.00%
IN_ST_TIM	4	0	0	0	0	4	100.00%
OUT_AMPM	4	0	0	0	0	4	100.00%
OUT_HRS0	4	0	0	0	0	4	100.00%
OUT_HRS1	4	0	0	0	0	4	100.00%
OUT_HRS2	4	0	0	0	0	4	100.00%
OUT_HRS3	4	0	0	0	0	4	100.00%
OUT_MINS0	4	0	0	0	0	4	100.00%
OUT_MINS1	4	0	0	0	0	4	100.00%
OUT_MINS2	4	0	0	0	0	4	100.00%
OUT_MINS3	4	0	0	0	0	4	100.00%
OUT_MINS4	4	0	0	0	0	4	100.00%
OUT_MINS5	4	0	0	0	0	4	100.00%
TIMER_CORE/COUNT1	938	0	0	14	0	952	100.00%
TIMER_CORE/STATE1	100	0	0	0	0	100	100.00%
TIMER_CORE	1038	0	0	14	0	1052	100.00%
TIMER_COMP	1128	0	0	14	0	1142	100.00%

If an output filename is not specified, the `create_test_vectors` command assigns a default name, `timer_comp.vdb`, to the vector database file. The `write_test` command takes this vector database and formats the vectors into the test program.

The test compiler produces two test programs: `timer_comp_schk.tdl`, which tests the scan circuitry, and `timer_comp_0.tdl`, which tests the combinational logic in the design. The `timer_comp_schk.tdl` program is shown in its entirety in Figure 7-20.

Figure 7-20. TDL'91 Scan Circuitry Test Program (timer_comp_schk.tdl)

```
(*=================================TITLE BLOCK=======================================*)
(*      LIBRARY TYPE:  TGC1000_COM_MAX.db TGC1000_TC.db                              *)
(*          CUSTOMER:  Katey Derbyshire of Texas Instruments Ltd.                    *)
(*    TI PART NUMBER:  *                                                             *)
(* PATTERN SET NAME:   TC_Syn_0                                                      *)
(* PATTERN SET TYPE:   SCANCHK                                                       *)
(*          REVISION:  1.00                                                          *)
(*          DATE:      09/22/92                                                      *)
(*===================================================================================*)

TDL_VERSION = "ASIC_TDL_91 1.0";

CONNECT P, VAR=( RESET_1, ST_TIME1, HRS_1, MINS_1, CLK_1
     TEST_SE_1, TEST_SI_1, HRS_OUT1_3, HRS_OUT1_2, HRS_OUT1_1_,
     HRS_OUT1_0_, MNS_OUT1_5_, MNS_OUT1_4_, MNS_OUT1_3_, MNS_OUT1_2_,
     MNS_OUT1_1_, MNS_OUT1_0_, AM_PM1, TEST_SO_1 ),
DEFPIN= ( IN 7, OUT 12 );
Date   : Tue Sep 22 11:16:59 1992

PERIOD = 100 NS;
CLOCK VAR=( CLK_1 ),
     HOLD0=45 NS, HOLD1=15 NS, PATTERN=010;

DELAY VAR=( RESET_1, ST_TIME1, HRS_1, MINS_1, TEST_SE_1, TEST_SI_1 ),
     OFFSET=5 NS;

STROBE VAR=( HRS_OUT1_3, HRS_OUT1_2, HRS_OUT1_1, HRS_OUT1_0_,
     MNS_OUT1_5_, MNS_OUT1_4_, MNS_OUT1_3_, MNS_OUT1_2_, MNS_OUT1_1_;
     OFFSET=95 NS;

PATH SC0, VAR=(TEST_SI_1, TEST_SO_1);

(*$ Synopsys Test Compiler, v2.2a-6762 (Feb 15, 1992) was used to genera
                  te this pattern set *)
(*$ INPUT VECTOR FILE = TIMER_COMP.vdb was the source file for this patt
                  ern set *)

SETR P:=T'YYYYLYYMMMMMMMMMMMMM;

SETR P:=T'LHLHLHHMMMMMMMMMMMMM;
SET  P:=T'LHLHCHHMMMMMMMMMMMMM;
SCAN FOR 19,
SCAN_IN  SC0:=T'HLLHHLLHHLLHHLLHHLL;
SCAN_OUT SC0:=T'MMMMMMMMMMMMMMMMMMMM';
END_SCAN;

SETR P:=T'LHLHLHYMMMMMMMMMMMM1;
SET  P:=T'LHLHCHYMMMMMMMMMMMMM;
SCAN FOR 19,
SCAN_IN  SC0:=T'YYYYYYYYYYYYYYYYYYY;
SCAN_OUT SC0:=T'0011001100110011000M';
END_SCAN;

END;
```

7.5.6 TDL'91

The TI ASIC Test Description Language '91 (TDL'91) is a new interface format for scan test patterns. The problem with many pattern formats is the large amount of redundancy in the test data. Input pins, apart from the *scan_in* pin, are held in their current state during the *scan_in* shift operations.

Similarly, output pins, apart from the *scan_out* pin, are masked during the *scan_out* shift operation. Scan test patterns in this format can require hundreds of megabytes of storage space and are thus expensive to store, transmit, and process.

TDL'91 has been developed primarily to minimize pattern file sizes by adding a few powerful constructs to remove the redundant information and reduce the file sizes into manageable proportions.

7.6 Copy Protection of Antifuse Architecture[†]

The unique design and antifuse technology of TI FPGA products offer inherent copy protection for your logic designs. After your design is programmed into a TI FPGA, it cannot be read back or deprocessed with accuracy.

7.6.1 Antifuses vs. Fuselinks

Most PLD devices employ a metal fuselink process that can be viewed easily under a medium-magnification optical microscope by removing the top. Once the programmed fuse locations required to generate a fuse map are determined, the name of the manufacturer known, and the rows and columns reordered, a PLD device can be copied, despite the presence of a programmed security fuse.

The TI antifuse is invisible to the die surface in either the programmed or unprogrammed state, because each fuse is sandwiched between the diffusion and poly layers. The actual programmed filaments are so small that deprocessing the die layer by layer to reach the antifuse will wash the filaments away. No amount of control during stripback is possible that will make all120K-750K fuses uniformly visible on the same die, which would take many iterations of units with the same progammed design to piece together even a part of the code.

7.6.2 Logic Probing or Voltage-Contrast Microscopy

A common but tedious and error-prone reverse-engineering technique is to decap a device and hand-probe the tracks, columns, and nodes; however, this technique will probably damage the layers of the device before completion.

A more sophisticated method involves toggling all the input pins slowly by utilizing a TEM or voltage-contrast microscope test system to determine the paths taken. Theoretically, the fuse map can be deciphered; however, it is practically impossible to achieve because of the complexity of the macros used, especially when using sequential logic.

[†] Contributed by Jim Ptasinsky, Engineering, Texas Instruments Incorporated.

7.6.3 Proprietary Fuse Addressing and Placement

ALS contains a proprietary place-and-route algorithm to determine the appropriate track and column paths needed and the fuses to be programmed; however, knowing which fuses are programmed is not enough to convert to a set of addresses, because the order in which the antifuses are programmed is unique, and it is this unique order that determines the address.

The only correct way to address a fuse depends on what is programmed previously on or adjacent to its particular track and in what order.; thus, even if a fuse map is available, it cannot regenerate a `.fus` programming file. If antifuses are not programmed in the specified order from the inside out, islands of fuses will form that cannot be addressed correctly, causing the device to be incompletely programmed.

The fuse addressing code required to read a fuse is proprietary and unpublished information. None of the available programming equipment has the capability to generate a fuse map or verify which fuses are programmed. In addition, the serial-scan method, with its partitioning, circuit architecture, and multiple paths, is complex enough to baffle most reverse engineers.

Even if a readback setup is available, your design increases the difficulty of verification. During the readback method when the module fanout is more than one, the sensing current takes multiple paths that increase ambiguity and make verification difficult. Ambiguity increases exponentially as the number of modules with high fanout increases.

7.6.4 FPGA Security Fuses

You can program both the M fuse and P fuse to enhance the security of your design even more. The M fuse disables the internal microprobing of each output module. The P fuse disables further programming and current sensing of programmed fuses.

Chapter 8

Application Examples

This chapter contains TI FPGA design application examples describing the implementation of the following: numeric keypad and seven-segment display decoder, DRAM controller with synthesis, CPU, UART, JTAG, and pipelined multiplier circuit. The principles discussed using these design examples apply to the entire TI FPGA family, which shares common design tools and processes.

The design flow includes a CAE environment for capture and simulation of the design and TI-ALS for place and route and device programming.

Two points are emphasized regarding circuit design for TI FPGAs: selecting macros for high implementation efficiency and using the timer to establish maximum operating frequency and input-to-output delay.

Knowledge of the logic module structure in TI FPGAs and the ways in which the module is used to obtain various logical functions is a valuable asset; for example, judicious use of input negation has led to the efficient mapping of Boolean equation descriptions onto the TI FPGA architecture in terms of both device utilization and speed of operation.

8.1 Incorporating a Keypad and Display in an FPGA Design†

This section describes the design, implementation, and testing of a numeric keypad and seven-segment display decoder in an FPGA design. In addition, speed critical nets are discussed along with I/O assignment, timing analysis, back annotation, device programming, and device utilization.

The design example described in this section is implemented on a TPC1010 device in a 68-pin PLCC package; however, the same principles apply to the entire TI FPGA family.

8.1.1 Keypad Scanner Design

The keypad keys are arranged in connected rows and columns. Pressing a key establishes a low-impedance path between the intersecting row and column connections beneath the pressed key. To use the keypad, scan each column sequentially and test each row for a low-impedance connection path.

The design example uses a 3-bit ring counter that scans the keypad by applying a logic 1 to each column in turn. To minimize the required circuitry, the ring counter scans the three columns rather than the four rows.

The ring counter is designed using Viewlogic Workview tools that provide schematic capture of the circuit, incorporation of elements from the TI FPGA macro library, and circuit simulation. In this case, three D-type flip-flops from the macro library are connected in cascade with Q outputs connected to the D inputs; the input clock to the circuit is buffered using the TI FPGA clock buffer. The circuit diagram for the keypad scanner is shown in Figure 8-1, along with the circuitry that initializes the ring counter with a single logic 1.

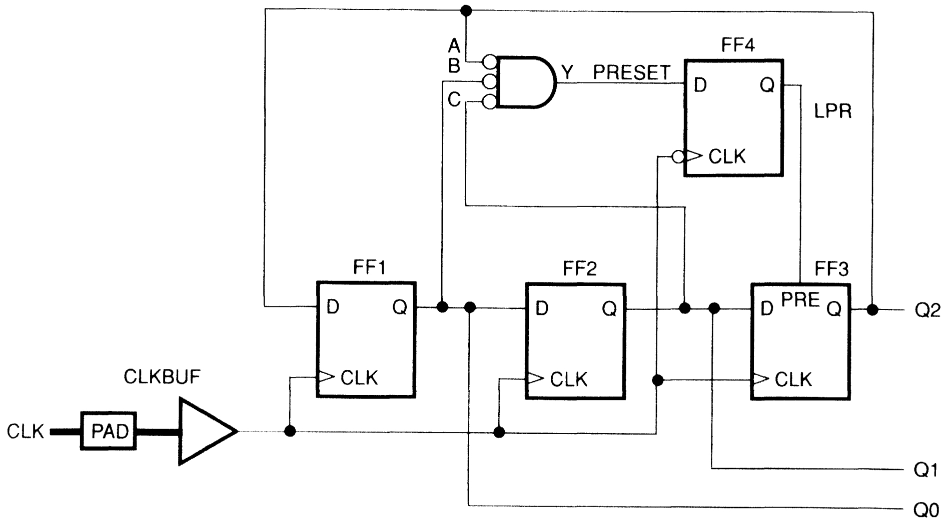
This synchronous initialization circuit will preset one of the flip-flops if all of the flip-flops are outputting logic 0. The preset signal is written as follows:

PRESET = !Q1 • !Q2 • !Q3

where *Q0*, *Q1*, and *Q2* represent the output of each of the three flip-flops and *!* indicates negation.

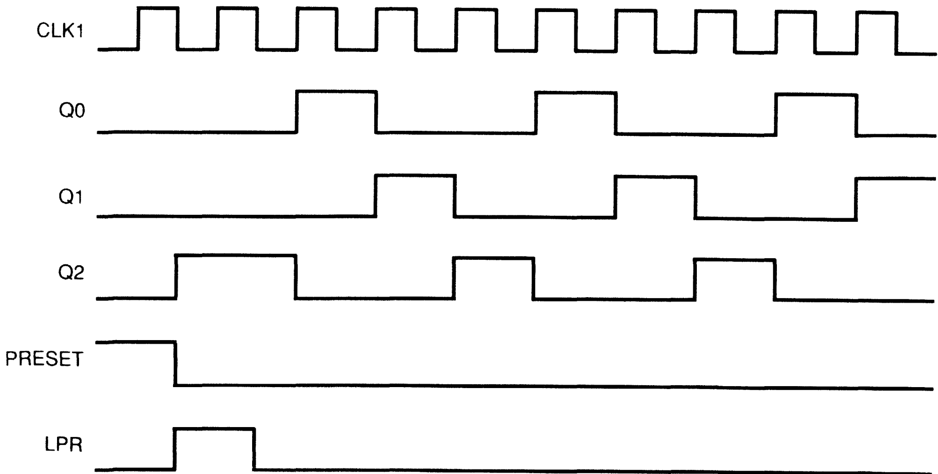
† Contributed by Patrick Naylor, Department of Electrical and Electronic Engineering, Imperial College, London.

Figure 8-1. Circuit Schematic of the Keypad Scanner



The action of the initialization circuit and the ring counter can be tested by using the Workview Viewsim circuit simulator to produce a timing diagram similar to the one shown in Figure 8-2.

Figure 8-2. Timing Diagram of the Keypad Scanning Circuit



Unit delays of 1 ns are used for all modules in a prelayout simulation such as this. These delays can be scaled from 1 ns to a typical 5.4 ns so that the Workview Viewsim circuit simulator will give realistic prelayout simulations. Back annotated postlayout simulations can also be obtained (see Section 8.1.3.8).

The circuit schematic for the keyboard scanner is designed using macros selected from the TPC10 macro library.

8.1.2 Seven-Segment Display Decoder Design

The Viewlogic tools used for schematic capture and simulation of the keypad scanning circuit described in Section 8.1.1 allow your design to work intuitively at the design stage and test ideas quickly on the simulator. Under certain circumstances, however, you may wish to adopt a formal approach based on the manipulation of Boolean equations. This section describes the decoder circuitry using this formal approach.

You must decode the keypad signals before you can determine which key is being pressed. The key-pressing process is conveniently described in terms of intermediate states; for example, state one indicates the button marked *1* is pressed, state two indicates the button marked *2* is pressed, etc.

There are seven keypad signals to decode: three column signals (Q0, Q1, Q2) and four row signals (D0, D1, D2, D3). The decoding operation should result in the decoding circuitry entering one of the twelve possible intermediate states: zero through nine, hash, and star.

A combinational circuit is required to carry out the decoding operation. Circuit complexity is reduced significantly by the design of the keypad scanning circuit, which requires Q0, Q1, and Q2 to be asserted one at a time. For example, if Q0 is a logic 1, it will always be true that Q1 and Q2 are both logic 0. Table 8-1 shows the Boolean equations written for the intermediate states.

Table 8-1. Boolean Equations for the Intermediate State

```
One    = Q0 •  D0 • !D1 • !D2 • !D3
Two    = Q1 •  D0 • !D1 • !D2 • !D3
Three  = Q2 •  D0 • !D1 • !D2 • !D3
Four   = Q0 • !D0 •  D1 • !D2 • !D3
Five   = Q1 • !D0 •  D1 • !D2 • !D3
Six    = Q2 • !D0 •  D1 • !D2 • !D3
Seven  = Q0 • !D0 • !D1 •  D2 • !D3
Eight  = Q1 • !D0 • !D1 •  D2 • !D3
Nine   = Q2 • !D0 •  D1 •  D2 • !D3
Star   = Q0 • !D0 • !D1 • !D2 •  D3
Zero   = Q1 • !D0 • !D1 • !D2 •  D3
Hash   = Q2 • !D0 • !D1 • !D2 •  D3
```

This simplifying step is one example of minimizing (or optimizing) Boolean equations. In general, you can use software tools to carry out the optimization. For example, ALES allows you to maximize logic utilization for speed and area in TI FPGAs and provides a synthesis tool for porting Boolean-equation or state-machine designs into the TI-ALS environment. ALES supports the following PLD design tools:

- ABEL
- CUPL
- PALASM
- LOG/IC
- PGADesigner

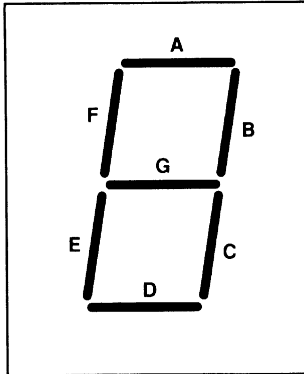
In addition, ALES accepts a PALASM 2 source file generated with a text editor or with ABEL, CUPL, or LOG/IC.

The remaining task in the design process requires further combinational circuitry and ensures that corresponding segments of the LED display illuminate when a key is pressed. To help design the circuitry, the truth table shown in Table 8-2 describes the logic levels applied to the individual segments of the display for each of the intermediate states, where A through G represent the seven segments of the LED display (see Figure 8-3).

Table 8-2. Truth Table

	A	**B**	**C**	**D**	**E**	**F**	**G**
One	0	1	1	0	0	0	0
Two	1	1	0	1	1	0	1
Three	1	1	1	1	0	0	1
Four	0	1	1	0	0	1	1
Five	1	0	1	1	0	1	1
Six	1	0	1	1	1	1	1
Seven	1	1	1	0	0	0	0
Eight	1	1	1	1	1	1	1
Nine	1	1	1	0	0	1	1
Star	0	1	1	0	1	1	1
Zero	1	1	1	1	1	1	0
Hash	0	1	1	0	1	1	0

Figure 8-3. LED Seven-Segment Display



These Boolean equations, derived from Table 8-2, are the complements of the output variables and provide a more efficient Boolean description:

!A = one + four + star + hash
!B = five + six
!C = two
!D = one + four + seven + nine + star + hash
!E = one + three + four + five + seven + nine
!F = one + two + three + seven
!G = one + seven + zero + hash

Combining the two sets of Boolean equations above yields those shown in Table 8-3, which represent the design of the display decoder.

Table 8-3. Boolean Equations for the Design of the Display Decoder

!B =	!D0 • D1 • !D2 • !D3 • Q2	+ Q1 • !D0 • D1 • !D2 • !D3
!C =	Q1 • D0 • !D1 • !D2 • !D3	
!D =	!D0 • !D1 • D2 • !D3 • Q2	+ !D0 • !D1 • !D2 • D3 • Q2
	+ Q0 • D0 • !D1 • !D2 • !D3	+ Q0 • !D0 • D1 • !D2 • !D3
	+ Q0 • !D0 • !D1 • D2 • !D3	+ Q0 • !D0 • !D1 • !D2 • D3
!E =	!D0 • D1 • !D2 • !D3 • Q1	+ D0 • !D1 • !D2 • !D3 • Q2
	+ !D0 • !D1 • D2 • !D3 • Q2	+ Q0 • D0 • !D1 • !D2 • !D3
	+ Q0 • !D0 • D1 • !D2 • !D3	+ Q0 • !D0 • !D1 • D2 • !D3
!F =	Q0 • !D0 • !D1 • D2 • !D3	+ D0 • !D1 • !D2 • !D3 • Q2
	+ D0 • !D1 • !D2 • !D3 • Q1	+ Q0 • D0 • !D1 • !D2 • !D3
!G =	Q0 • D0 • !D1 • !D2 • !D3	+ Q0 • !D0 • !D1 • D2 • !D3
	+ !D0 • !D1 • !D2 • D3 • Q2	+ !D0 • !D1 • !D2 • D3 • Q1

8.1.3 Keyboard Scanner and Decoder Implementation

The TI FPGA macro library contains useful circuit elements made up of hard macros, such as gates, buffers, flip-flops, and latches; and soft macros, such as counters, decoders, comparators, multiplexers, registers, multipliers, shift registers, TTL equivalents, and a UART.

For efficient device utilization and minimum delays, the shift register required to make the ring counter should be built from three individual flip-flops rather than a library part.

For example, the SREG4 macro, a 4-bit shift register with clear, best fits the shift register design but uses eight logic modules compared to the six required by the three flip-flops; thus, the three-flip-flop design is more efficient.

Circuit delays are estimated on first approximation using the formula:

5.4 ns • number of logic levels

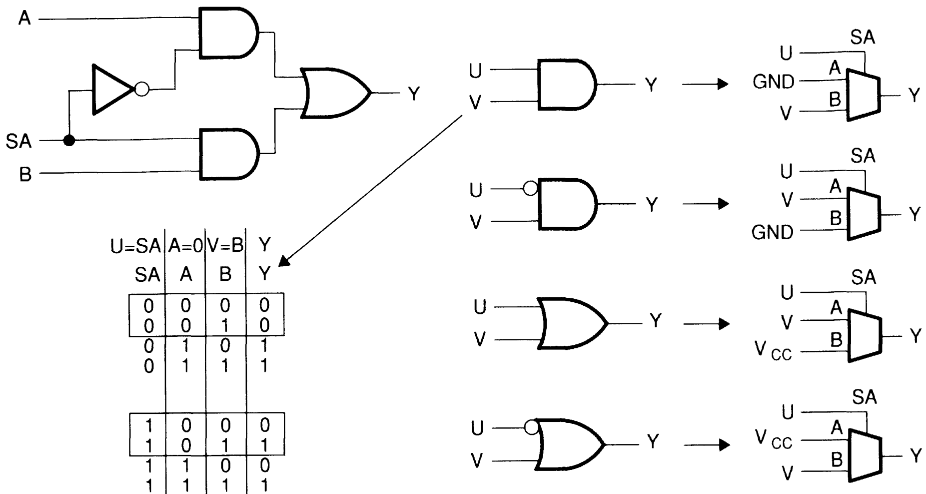
The three-flip-flop design uses a single logic level rather than the two logic levels used by the SREG4 and thus inserts only half the delay, as indicated by the above formula.

The Boolean equations in Section 8.1.2 describe the combinational circuits required to decode the outputs from the keypad scanner. Implementing these equations directly using a combination of AND gates, OR gates, and inverters is usually inefficient because of the lack of available logic macros and the architecture of the logic module. For example, the Boolean equations require five-input NAND gates that are unavailable in the library and must be constructed from two-, three-, and four-input gates.

8.1.3.1 Implementing Gates With Free Input Negation From 2-Input Multiplexers

Figure 8-4 shows an element of the TI FPGA logic module in the form of a two-input multiplexer.

Figure 8-4. Obtaining Logic Functions From the TI FPGA Logic Module



SA	A	B	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

NOTES: 1. FPGA hard macros are built from logic modules by programming the appropriate antifuses to connect the module inputs to GND or V_{CC}.

2. The logic module architecture is capable of building complex gates or flip-flops with a minimum of resources.

The truth table in Figure 8-4 indicates how output Y corresponding to the AND and OR function of the two inputs can be obtained from the multiplexer circuit. The figure shows that the same logic module can produce AND or OR functions of two inputs when one input has an additional inversion. This additional input inversion requires no additional resources because the logic module design employed in TI FPGAs is able to implement input inversion with a simple modification of the input multiplexer's program. For example, consider the equations

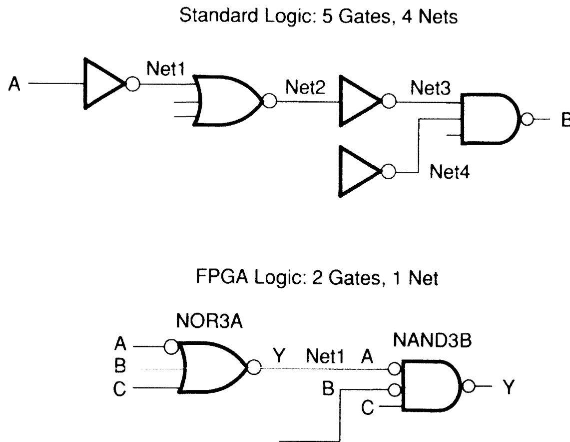
Y = U.V
Y = !U.V

In Figure 8-4, the lines of the multiplexer truth table enclosed in boxes make up the truth table for the AND function. These lines are selected by setting *A* to logic 0; the two input signals are fed to *SA* and *B* in order to implement the first of the two-example function.

If implementing the second of the two example functions is required (i.e., AND with one input inversion), set *B* to logic 0 and feed the two input signals to *SA* and *A* so that no extra resources are required for the input inversion.

The property of free-input inversions to gates described above increases implementation efficiency and reduces the number of gate propagation delays. For example, Figure 8-5 shows how to reduce a five-gate circuit to a two-gate circuit. When gate propagation delays limit the maximum operating frequency of a circuit, input inversions can be a powerful technique for speed enhancement.

Figure 8-5. Circuit Reduction Using Input Inversions



8.1.3.2 Implementing the Keypad Decoder Using TI FPGA Macros

The Boolean equations describing the required circuitry of the keypad decoder are broken down into elements that can be efficiently implemented using TI FPGA macros. Close attention has been paid to the use of input negation to improve efficiency.

Boolean terms occurring more than once must be generated only once and then referred to as intermediate terms *I0, I1, I2,* etc., as in the following example:

I0 = D0 • !D1 • !D2 • !D3
I1 = !D0 • D1 • !D2 • !D3
I2 = !D0 • !D1 • D2 • !D3
I3 = !D0 • !D1 • !D2 • D3

Refer to the *FPGA Macro Library Summary* to see that the macro AND4C will implement any of the intermediate terms completely, including the necessary inversions.

You can rewrite the Boolean equations based on the intermediate terms. As Table 8-4 shows, like terms in the equations are vertically aligned.

Table 8-4. Boolean Equations Based on Intermediate Terms

!A =	I3•Q2 +	I0•Q0 +	I1•Q0 +	I3•Q0							
!B =					I1•Q2 +	I1•Q1					
!C =							I0•Q1				
!D =	I3•Q2 +	I0•Q0 +	I1•Q0 +	I3•Q0 +				I2•Q2 +	I2•Q0		
!E =		I0•Q0 +	I1•Q0 +			I1•Q1 +		I2•Q2 +	I2•Q0 +	I0•Q2	
!F =		I0•Q0 +					I0•Q1 +		I2•Q0 +	I0•Q2	
!G =	I3•Q2 +	I0•Q0 +							I2•Q0 +		I3•Q1

Because like terms have to be generated only once, you save circuitry. The AND2 macro is suitable for implementing the 2-input AND function.

Terms on the left side of the Boolean equations in Table 8-4 are complements; thus NOR functions have to be implemented. The *!C* equation requires only an inversion that can be obtained implicitly by using a NAND gate instead of the AND gate previously indicated. The 2-input NAND2 macro is available in the library.

A consequence of using the inversion is that it must be considered wherever it occurs in other equations. In this case, the only other equation that uses the *I0•Q1* term is the *!F* equation.

The *!B* equation requires a 2-input NOR gate represented in the library by the NOR2 macro.

The *!A* and *!G* equations require the 4-input NOR function obtained from the library by the macro NOR4; however, the NOR4 requires two logic modules and uses twice the resources of a 1-logic-module macro, thus reducing the quantity of chip resources available for implementing other circuitry.

The NOR4A macro available from the library is also a 4-input NOR gate with one inverted input; however, unlike the NOR4 macro, the NOR4A requires only one logic module. In order to utilize the NOR4A for the design example, an input term to the macro must be complemented in combination with the input negation in order to form a double complement that cancels out.

Because the *I0 • Q0* term is common to the *!A* and *!G* equations, the double complement can be applied. In the design example, *!(I0•Q0)* is generated using the NAND2 macro. Because the *I0 • Q0* term is included in the *!D*, *!E*, and *!F* equations, the inversion must be considered when these equations are implemented.

First, rewrite all the equations, including the additional inversions and moving the overall inversion in each equation from the left side to the right side. Expressions are thus given for each segment of the display rather than for its complement (Table 8-5).

Table 8-5. Boolean Equations With the Additional Inversions

A = !(I3•Q2 + !(!I0•Q0)) + I1•Q0 + I3•Q0)
B = !(I1•Q2 + I1•Q1)
C = !(!(!I0•Q1)))
D = !(I3•Q2 + !(!(I0•Q0)) + I1•Q0 + I3•Q0 + I2•Q2 + I2•Q0)
E = !(!(!(I0•Q0)) + I1•Q0 + I1•Q1 + I2•Q2 + I2•Q0 + I0•Q2)
F = !(!(!(I0 •Q0)) + !(!(I0•Q1)) + I2•Q0 + I0•Q2)
G = !(I3•Q2 + !(!I0•Q0)) + I2•Q0 + I3•Q1)

The *F* equation has two input terms inverted and double complements inserted to maintain the integrity of the equation. The NOR4B macro, which has two inverted inputs (one for each of the inverted product terms) and requires only one logic module, is used to implement the expression for *F*.

The *!D* and *!E* equations that remain to be implemented both contain six terms, one of which has been subjected to an inversion during the above procedures. A combination of two gates is required to implement the equations, which also contain the common terms *I1 • Q0*, *I2 • Q2*, and *I2 • Q0*.

Because efficiency dictates that common terms be implemented only once, the 3-input OR3 macro can be used to generate the logic OR of the common terms.

In addition, the NOR of the output of the OR3 macro must be formed with three other terms, one of which has an additional inversion, to implement the *D* and *E* equations. Two NOR4A macros can be used in conjunction with a single OR3, one to generate *D* and one to generate *E*.

Figure 8-6 shows the circuit schematic for the generation of the intermediate terms *I0* to *I3*.

Figure 8-6. Circuit Schematic for Generating the Intermediate Terms

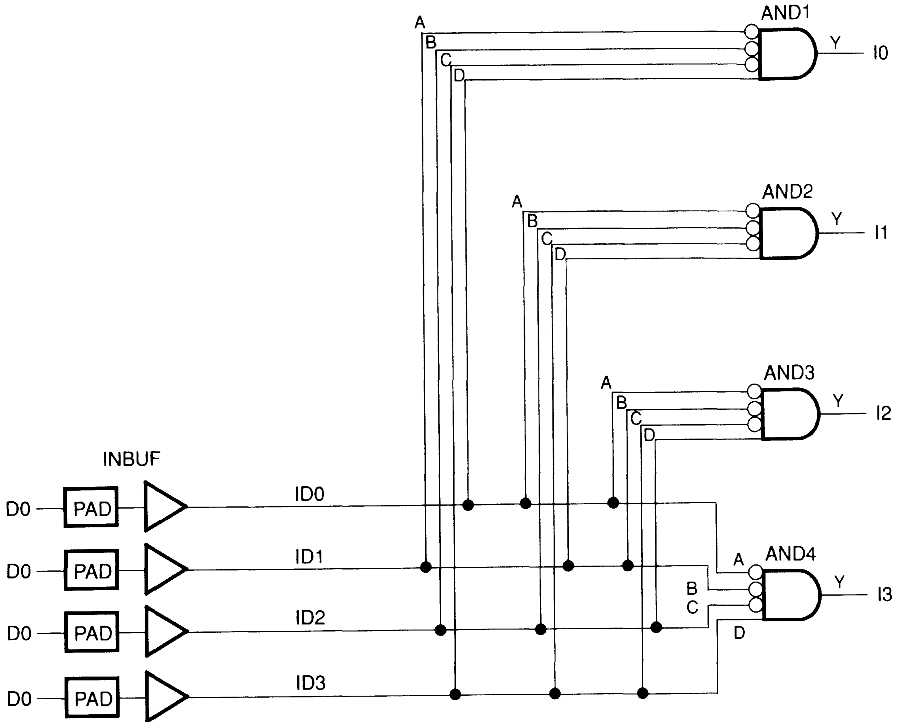
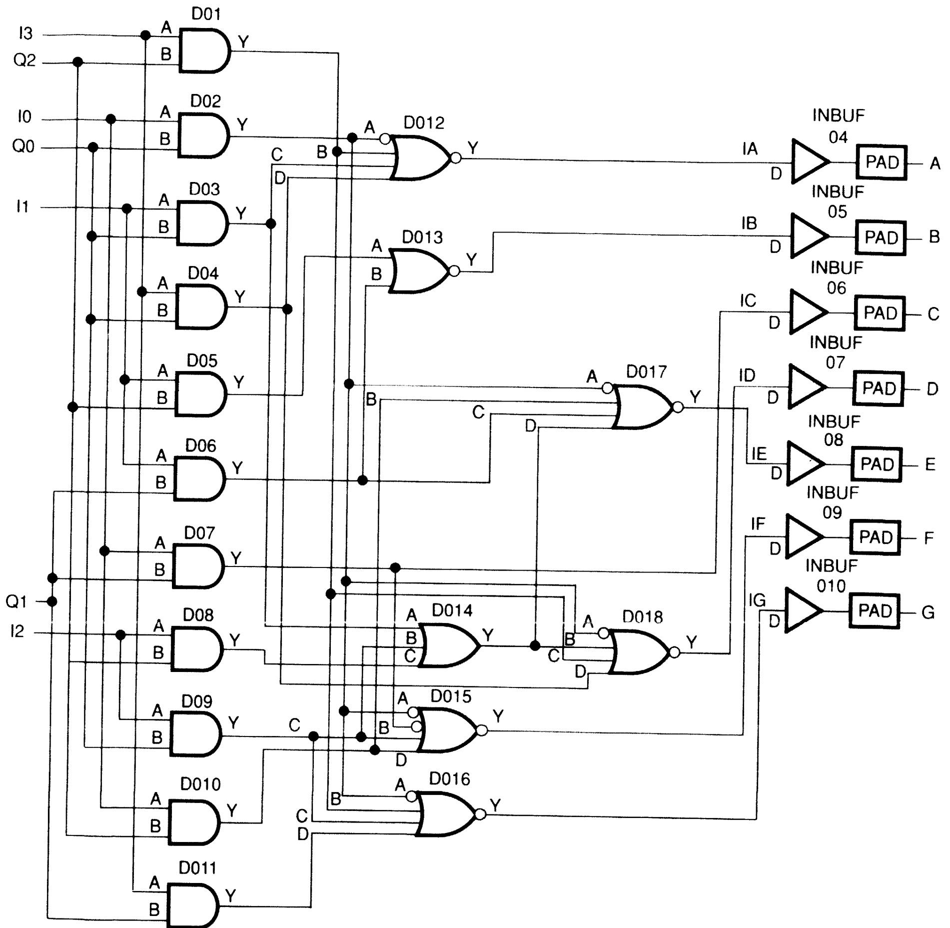


Figure 8-7 shows the circuit schematic for the rest of the keypad decoder.

Figure 8-7. Circuit Schematic for the Keypad Decoder



8.1.3.3 From Schematic to Silicon Using TI-ALS

An important step in progressing from a schematic representation of the circuit to a representation suitable for down-loading to silicon is to export a wire list generated in the CAE environment to TI-ALS. TI-ALS is a powerful tool that allows you not only to program silicon but also to iterate a design in order to optimize any given criterion. More often than not, the criteria of interest are speed of operation and device utilization efficiency.

TI-ALS builds a suite of design files in ASCII text format. Each file contains a specific type of information and is identified by one of these specific filename extensions:

- `.adl` (exported netlist file)
- `.crt` (critical path file)
- `.ipf` (pin editor data file)
- `.def` (definition file)
- `.cob` (validator combine file)
- `.val` (validation passed file)
- `.vld` (validation log file)
- `.pin` (pin assignment file)
- `.dfr` (design for routability file)
- `.loc` (macro location file)
- `.pli` (placement information file)
- `.seg` (track segment file)
- `.rti` (routing information file)
- `.del` (physical delay file)
- `.map` (macro placement file)
- `.fus` (fuse file)
- `.dtb` (workview delay table file)

See the following sections for more discussion on the use of these files.

8.1.3.4 Net Criticality

The speed of certain nets can be critical to circuit operation. TI-ALS allows the design to mark speed-critical nets based on the following criticality levels:

- Fast

  Up to six percent of the total nets can be marked fast-critical, which instructs the routing algorithm to use the shortest, fastest track layout possible

- Medium

  Up to 15 percent of the total nets can be marked medium-critical, which instructs the routing algorithm to avoid using long vertical or horizontal tracks. Uncritical and default nets are not given special consideration by the router.

- Uncritical

  Uncritical nets are not given special consideration by the router.

- Default

  Default nets are not given special consideration by the router.

A net is marked critical by editing the `.crt` file, which is initially blank but can be edited using almost any text editor according to the defined syntax. After the editing process, you will need to invoke the **`certify`** command on the file to update the file control information.

8.1.3.5 Pin Assignment

The relative location of I/O pins to other parts of the internal circuitry affects operating speed and device utilization. I/O placement is important where it is necessary to use parts of pre-existing circuitry to bring out particular signals from the device to particular locations.

Pin assignment information is stored in the `.ipf` and `.pin` design files. Both automatic and manual pin assignment are available using TI-ALS. In the automatic method, the TI-ALS software lays out the I/Os and assigns pins automatically. All elements making up the design perform to an equal specification; i.e., the TI-ALS software attempts to optimize such criteria as minimum delay and maximum utilization.

In cases where a part of the circuit may be critical to the overall performance of the circuit design, I/Os can be assigned manually within TI-ALS so that the performance of the critical element is optimized. For example, you can eliminate some routing delays from a speed critical path using a combination of the `.loc` design file and manual pin assignment.

Manual pin assignment is also useful in a TI FPGA design to be incorporated in a design containing pre-existing elements. For example, manual pin assignment brings out particular signals from the device to particular locations on a circuit board; however, additional routing may cause speed degradation.

The design example requires seven I/Os to be connected between the keypad and the FPGA and seven between the FPGA and the display. Pull-down resistors must be used for the four D-input I/Os (D0, D1, D2 and D3) in order for the scanner circuitry to function correctly. Manual pin assignment can keep the four inputs close together so that an in-line resistor module can be used as pull-down resistors.

8.1.3.6 *Timing Analysis and Maximum Operating Frequency*

The TI-ALS software incorporates a comprehensive static timing analysis providing postlayout path-delay information under a variety of predefined conditions, the most common of which is the worst-case condition.

To investigate the maximum frequency at which the keyboard can be scanned, use the timer to determine the maximum operating frequency of the keyboard scanning circuit.

To determine the maximum operating frequency of this or any other synchronous circuit, first find the worst-case delay between clock transitions and data becoming valid on the output of latches or flip-flops. When calculating these delays, include any gate delays and set-up times in the paths. Delays determined in this way are called register-to-register delays.

The TI-ALS timer operates by finding path delays between specified start and end points in the circuit. The set of start points is called the *startset*; the set of end points is called the *endset*.

For convenience, TI-ALS maintains default sets of start and end points. Two of the default sets are called *CLOCK*, containing the set of all clock inputs, and *GATED*, containing the set of all gated outputs.

To find the maximum operating frequency of the circuit, determine the longest delay between any member of the *CLOCK* set and any member of the *GATED* set using the following two-step procedure:

1) Using the `Set` command from the Timer menu, select *startset* as CLOCK and *endset* as GATED.

2) Using the `Longest` command from the Timer menu, determine the longest delay between these two sets.

The four longest delay paths determined by the `Longest` command are shown in Table 8-6.

Table 8-6. First Longest Path to All Endpins

Rank	Total	Start Pin	First Net	End Net	End Pin
0	16.2	FF4CLK	LPR	PRESET	FF4:D
1	12.7	FF4:CLK	LPR	Q2	FF1:D
2	10.1	FF1:CLK	Q0	Q0	FF2:D
3	6.8	FF2:CLK	Q1	Q1	FF3:D

As Table 8-6 shows, the worst delay path occurs between the CLK input PRESET net.

Using the `Expand` command from the Timer menu, you can look with more detail at the sources of delay in any given delay path. For example, expanding the path of Rank 0 provides the information shown in Table 8-7.

Table 8-7. First Longest Path to FF4:D (Rising) (Rank: 0)

Total	Delay	Typ	Load	Macro	Start Pin	Net Name
16.2	2.6	T_{su}	0	DF1B	FF4:D	
13.6	3.0	T_{pd}	1	AND3C	AND0:A	PRESET
10.6	6.3	T_{pd}	7	DFP1	FF3:PRE	Q2
4.3	4.3	T_{cq}	4	DF1B	FF4:CLK	LPR
0.0	0.0	P_{sk}	8		FF4:CL	CLK1

The five delay elements listed in Table 8-7 contribute to the total delay of 16.2 ns. The FF4 flip-flop has both a required setup time (T_{su}) of 2.6 ns and a clock-to-Q delay (T_{cq}) of 4.3 ns. The propagation delay (T_{pd}) from the input of the AND0 macro and along the PRESET net is listed as 3 ns. The propagation delay through the asynchronous preset input of the FF3 flip-flop and along the net Q2 is listed as 6.3 ns. The sum of these delays is 16.2 ns.

Having determined the worst-case register-to-register delay path, you can calculate the maximum operating frequency F_{max} as follows:

$$F_{max} = \frac{1}{\text{worst} - \text{case delay}}$$

$$= \frac{1}{16.2 \times 10^{-9}}$$

$$= 1.7 \quad \text{MHz}$$

8.1.3.7 Input to Output Delay

The TI-ALS software can evaluate the input-to-output delay of a combinational circuit, which is useful for various reasons; e.g., checking that the setup times of any following sequential circuit elements are satisfied.

To evaluate the input-to-output delay of a combinational circuit, find the longest delay between any input and output pin by using two further predefined sets of pins, INPAD and OUTPAD. INPAD contains the set of all input pads; OUTPAD contains the set of all output pads. Use the following two-step procedure to determine the maximum operating delay:

1) Using the `Set` command from the Timer menu, select *startset* as INPAD and *endset* as OUTPAD.

2) Using the `Longest` command from the Timer menu, determine the longest delay between the two sets.

Table 8-8 shows the following results for worst-case conditions.

Table 8-8. First Longest Path to All Endpins

Rank	Total	Start Pin	First Net	End Net	End Pin
0	67.7	G0:PAD	ID0	E	G8:PAD
1	66.4	G0:PAD	ID0	D	G7:PAD
2	64.0	G3:PAD	ID3	G	G10:PAD
3	63.9	G3:PAD	ID3	F	G9:PAD
4	62.8	G3:PAD	ID3	A	G4:PAD
5	53.5	G0:PAD	ID0	B	G5:PAD
6	52.8	G3:PAD	ID3	C	G6:PAD

Table 8-8 shows the longest delay path from the G0:PAD (D0 signal input) to the G8:PAD (E segment output) to be 67.7 ns. The timing analysis informs you of the delay between a valid D0 signal being input to the decoder and the output signals to the display becoming valid. For example, if the segment outputs are to be latched by either an internal or external 7-bit latch, appropriate timing for the gate input to the latches can be determined.

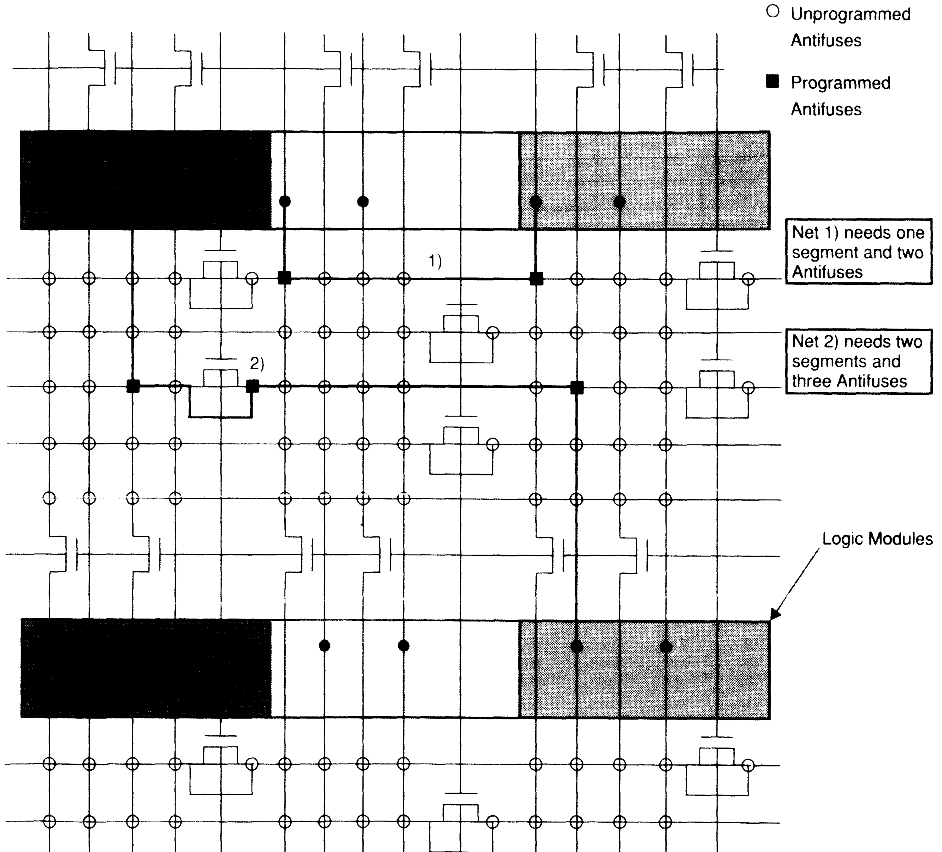
8.1.3.8 Back Annotation

The `.dtb` TI-ALS design file, generated during place-and-route operations of the TI-ALS design process, contains post-layout routing delay information fed back to the Workview Viewsim simulator to back annotate the results of circuit simulation with actual device delays. This feature of the TI FPGA design process allows you to carry out not only functional simulation but also device simulation.

8.1.3.9 Device Programming

TI FPGA device programming, supported by the TI-ALS software and the Activator programming unit, is achieved by blowing blow-to-make antifuses using a programming voltage pulse (Figure 8-8).

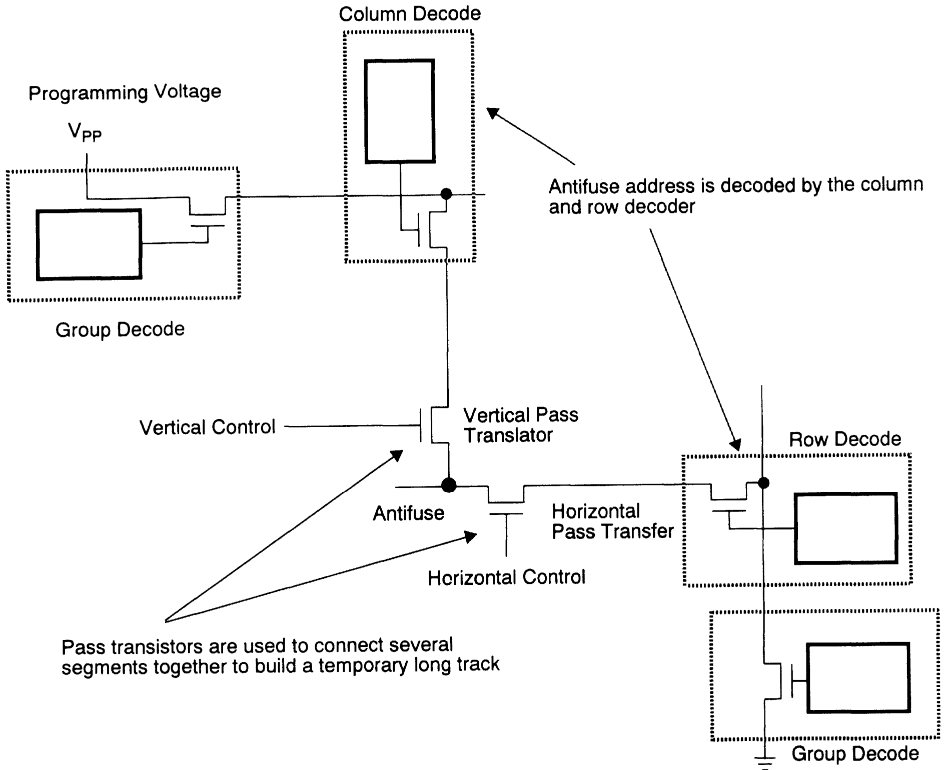
Figure 8-8. Route Programming



Because the device is initially full of nonconnecting logic modules, it is normally impossible to address logic modules other than those outside of the device; however, TI FPGA makes use of pass-transistors to perform routing during programming.

Vertical and horizontal pass transistors provided on the device allow a temporary connection between the programming pins on the device and any antifuse (Figure 8-9).

Figure 8-9. Pass Transistors Used for Programming



8.1.3.10 Device Utilization

Device utilization is reported by the TI-ALS software in several TI-ALS design files, such as the .vld file, which contains a summary of the information. See Figure 8-10 for a summary of information for this design.

Figure 8-10. TI-ALS .vld File

```
VLD5300: Design uses:
    31 logic modules
    12 io modules
    1 clock module
92 module inputs driven by 35 signal nets with 2.62 average fanout.
```

The example 68-pin PLCC TPC1010 device contains 295 logic modules and 57 I/Os, which indicates that the implementation has used approximately 10 percent of the logic modules and about 20 percent of the I/Os.

8.1.3.11 Debugging and Testing

After a device is programmed, debugging and testing may be necessary. The TI-ALS supports two such operations. One is a functional debug during which the Activator programming unit exercises the device under control of the host PC. The inputs of the device under test can be controlled using either test vectors or a control language, and the outputs of the device can be monitored and reported to you. Functional debug allows you to view the actual function of the device rather than a simulation.

In-circuit debug using Actionprobe diagnostic tools offers an optional debugging and testing operation. This unique feature of the TI FPGA family lets you observe all internal nets using, for example, a cathode ray oscilloscope (CRO), while the device operates in-circuit. Up to two signal may be viewed simultaneously.

For details on the functional debugging control language and use of Actionprobe tools, see TI-ALS documentation.

8.2 Implementing a DRAM Controller†

As microprocessors increase in number and complexity, interfacing to the system memory and interface chips becomes more of a problem. Additional circuitry is often required to handle address multiplexers, timing relationships, and, particularly for DRAMS, refresh control signals. This section shows you how implementing a simple DRAM controller in an antifuse-based TI FPGA can make your design more flexible. You are introduced to basic DRAM control along with the various design flow options available for implementation in an FPGA.

In addition, this section discusses how the TI design environment can enhance your FPGA design by minimizing design and development time. The example design is implemented in a TPC1010A device, which is the smallest member of the TPC10 series family with a capacity of 295 logic modules.

Unlike the SRAM, the memory cell of the DRAM loses its charge along with its contents unless the charge is updated by a refresh signal. The refresh rate necessary depends on the type of DRAM but is typically in the order of a millisecond. Memory refresh must be coordinated with the continuing memory read-and-write cycles that write new data into the DRAM and read data from it. Some arbitration must thus be carried out by the controller in conjunction with the system CPU.

Because each system typically has its own unique timing and interfacing requirements, using an FPGA to create a customized controller provides you with a flexible design and the rapid time to market you need.

8.2.1 Functional Description

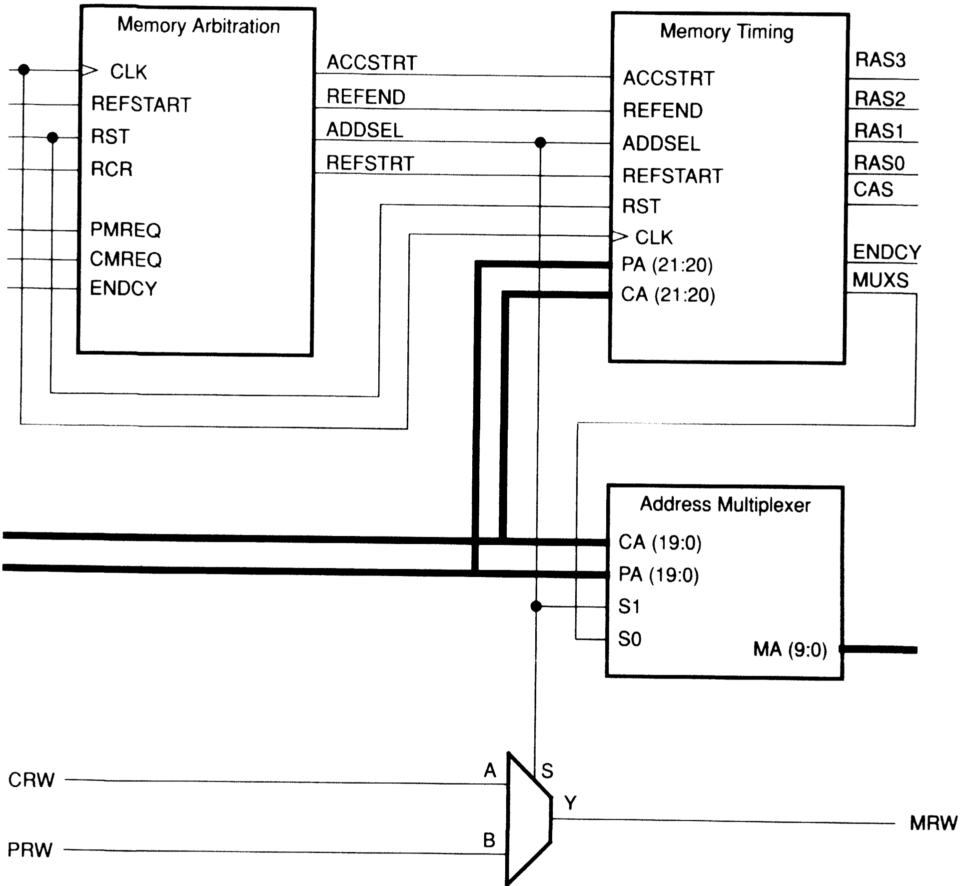
This DRAM controller performs three basic functions:

1) Memory arbitration between data accesses to and from memory and memory refresh.
2) Refresh signal generation and general timing and control to indicate to the arbitration circuit that a refresh is required and to set the address and refresh signals required for a refresh cycle.
3) Address multiplexing providing the DRAM with required row, column, and page address.

No internal latching of addresses is carried out on the chip, although it is recommended if needed to prevent the use of external latches on the board. Figure 8-11 shows the DRAM controller block diagram.

† Contributed by Mohan Maheswaran, Technical Marketing, Texas Instruments Ltd.

Figure 8-11. DRAM Controller Block Diagram



8.2.2 Memory Arbitration

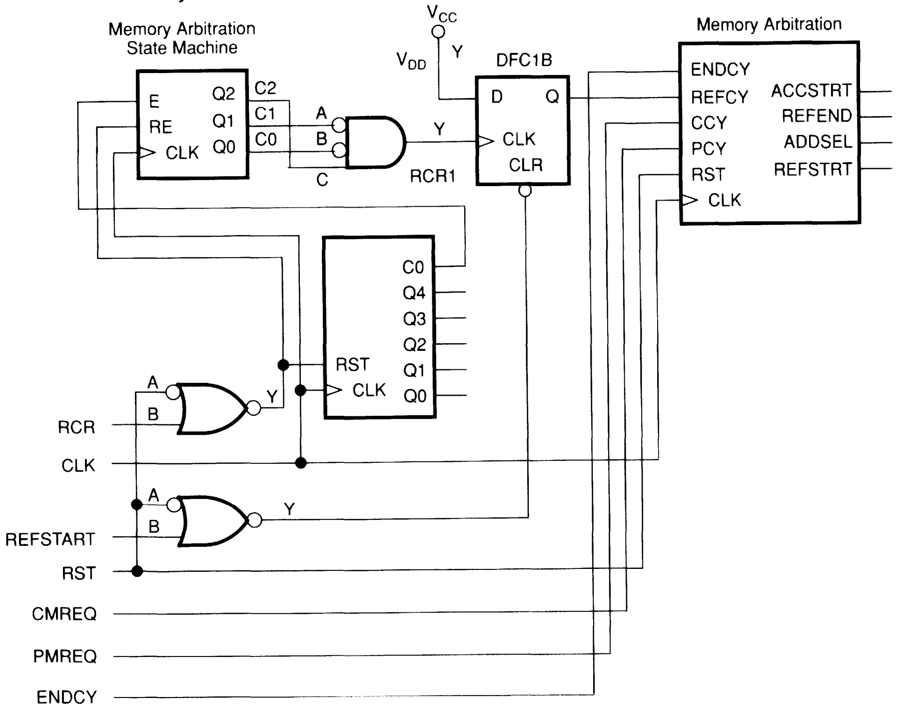
The memory arbitration block provides signals to the other functional blocks that indicate:

- The start and end of a refresh cycle
- The signal control necessary for selection of addresses
- The signals necessary to signify the memory can be accessed

The block provides the signals described based on the system inputs it receives, which take the form of a column/page request and refresh control that can be decoded from the CPU control signals.

In addition, a system clock and reset signal is provided by three state machines, as shown in Figure 8-12. This is by no means the optimum approach for such a function but it helps demonstrate the flexible approach that one can take when designing with FPGAs. The state machines themselves can be designed in several ways and two approaches are shown later on.

Figure 8-12. Memory Arbitration Unit Block Diagram

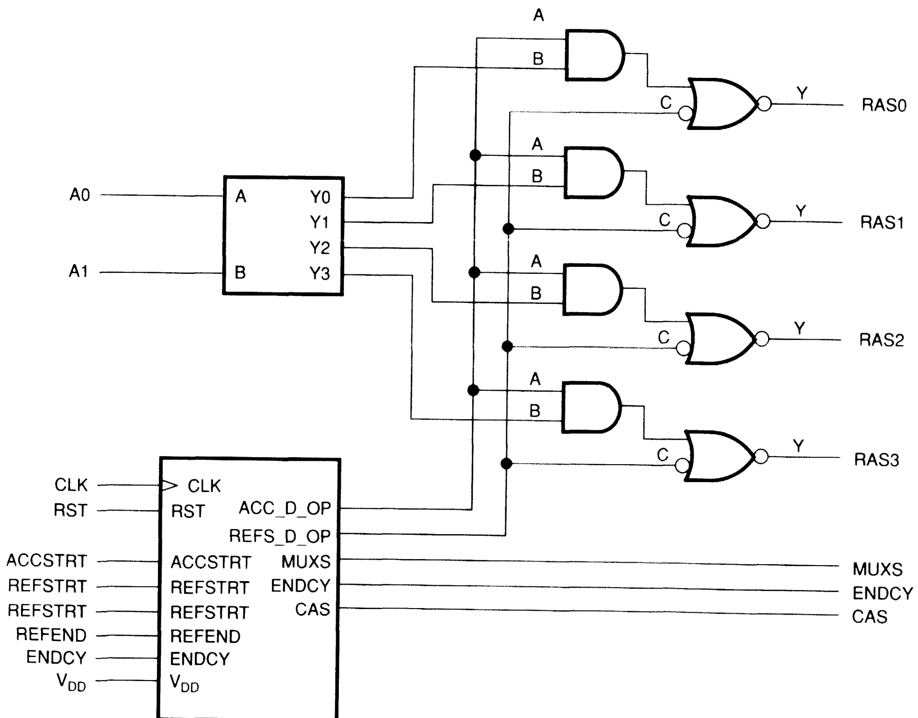


8.2.3 Refresh Rate Generator and Timing Control

The refresh rate generator and timing control block provides the required row and column refresh strobes and, together with the memory arbitration block, controls the selection of addresses to be output onto the address bus via the multiplexer unit (Figure 8-13).

This block also signifies the end of the refresh cycle to the other blocks within the DRAM controller. Although the design of the block depends on the system requirements as with any other block, modifying the basic structure of a controller to design a customized controller is not a significant task. Inputs to this unit are the outputs of the memory arbitration unit and two addresses decoded into four row strobe signals. A 2-to-4 decoder decodes two address inputs to provide four row strobes. These four strobes are used to latch the row address into a bank of DRAMs.

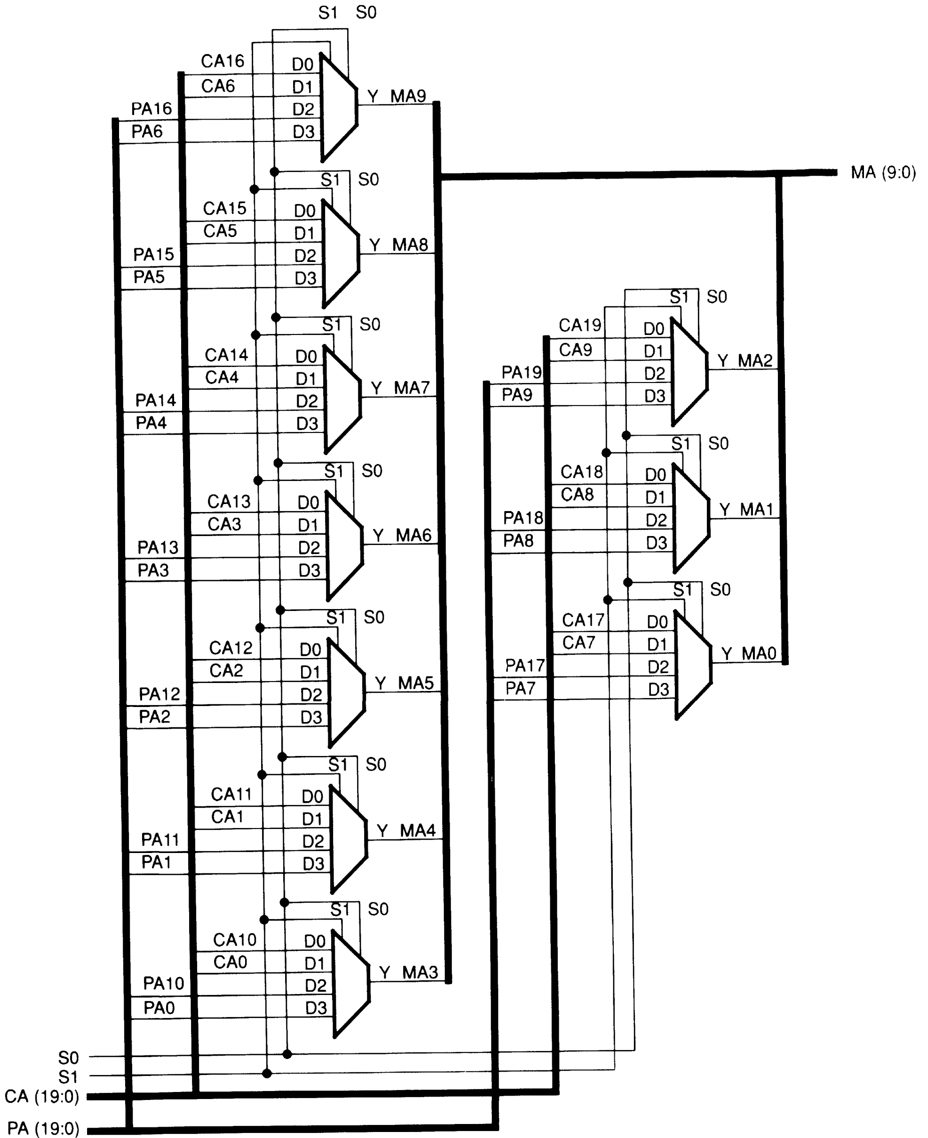
Figure 8-13. Refresh Rate Generator and Timing Control Block Diagram



8.2.4 Address Multiplexer

The address multiplexer block is the simplest to design (Figure 8-14).

Figure 8-14. Address Multiplexer Block Diagram



Block output is a 10-bit address bus allowing one Mbyte of memory to be addressed. The inputs to the block are two control inputs and two 20-bit buses. The multiplexer selects the correct address bits depending on whether the controller is working in paged or normal mode. The bus size and the protocol associated with each system will be different but this example shows how large bus structures can be handled effectively in an FPGA.

8.2.5 DRAM Controller Design Flow

The design flow on the front end; that is, prior to place and route, is most effectively demonstrated by focusing on the implementation of one part of the complete design; e.g., a state machine in the memory arbitration unit.

8.2.6 Design Flow Using TI ProLogic Software

The TI proLogic compiler software provides one option for designing the state machine. The proLogic `.pld` file in Figure 8-15 show the high level of design of the state machine.

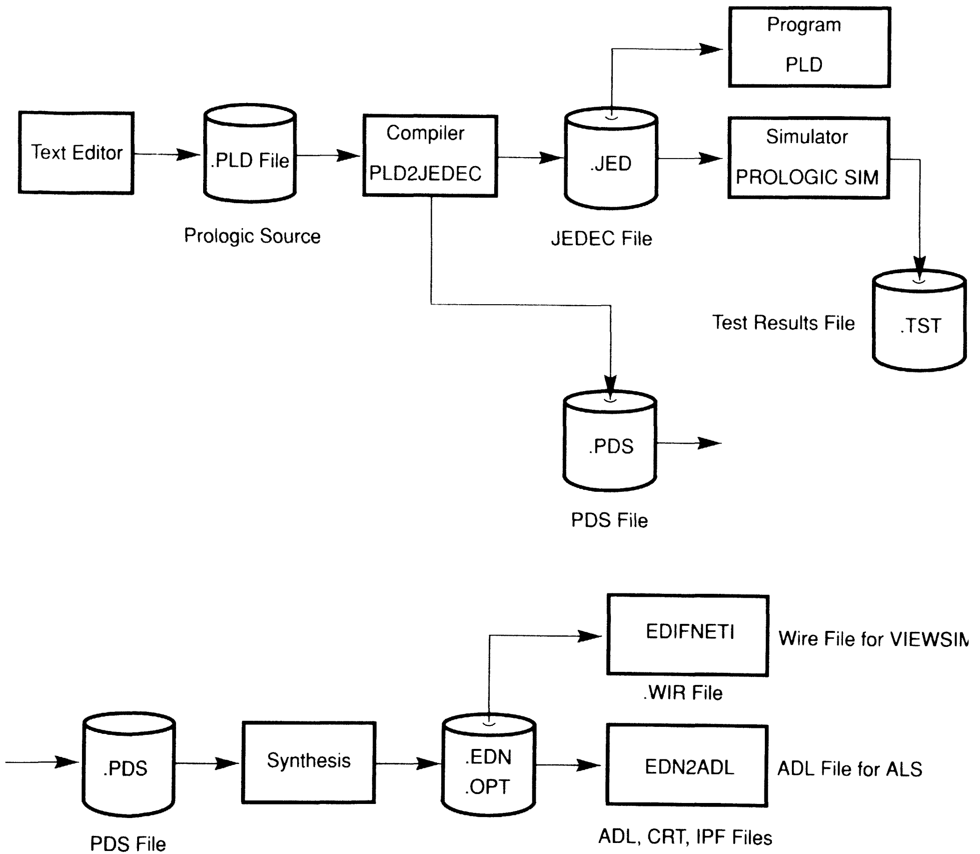
Figure 8-15. The `.pld` *Source File*

```
/*********************************************************/
/*      Author: Mohan Maheswaran                        */
/*     Language:ProLogic                   */
/*      Device 22V10                       */
/*     Purpose: DRAM Controller mem. arb. nit state machine */
include 022v10;
define CLK = pin1; /* DEFINE INPUTS */
define RE = !pin2;
define E = pin3;
define Q2 = pin20;
define Q1 = pin21;
define Q0 = pin22;
state_diagram Q2, Q1, Q0 {
if {RE}
    A;
state A=000
   if {E}
    B;
    else
    A;
state B=001
   if {E}
    C;
    else
    B;
state C=011
```

```
   if {E}
    D;
    else
    C;
state D=111
   if {E}
    F;
    else
    D;
state F=101
   if {E}
    G;
    else
    F;
state G=100
   if {E}
    H;
    else
    G;
state H=110
   if {E}
    I;
    else
    G;
state I=010
   if {E}
    A;
    else
    I;
}
/* active high */
Q2 = q;
Q1 = q;
Q0 = q;
Q2.oe = 1;
Q1.oe = 1;
Q0.oe = 1;
test vectors {
/*  CLK   RE     E    Q2    Q1    Q0*/
   pin1  pin2 pin3  pin20 pin21 pin22;
    0     0     0     L     L     L;
    C     0     X     L     L     L;
    C     1     0     L     L     L;
    C     1     1     L     L     H;
    C     1     1     L     H     H;
    C     1     1     H     H     H;
    C     1     1     H     L     H;
    C     1     1     H     L     L;
    C     1     1     H     H     L;
    C     1     1     L     H     L;
    C     1     1     L     L     L;
```

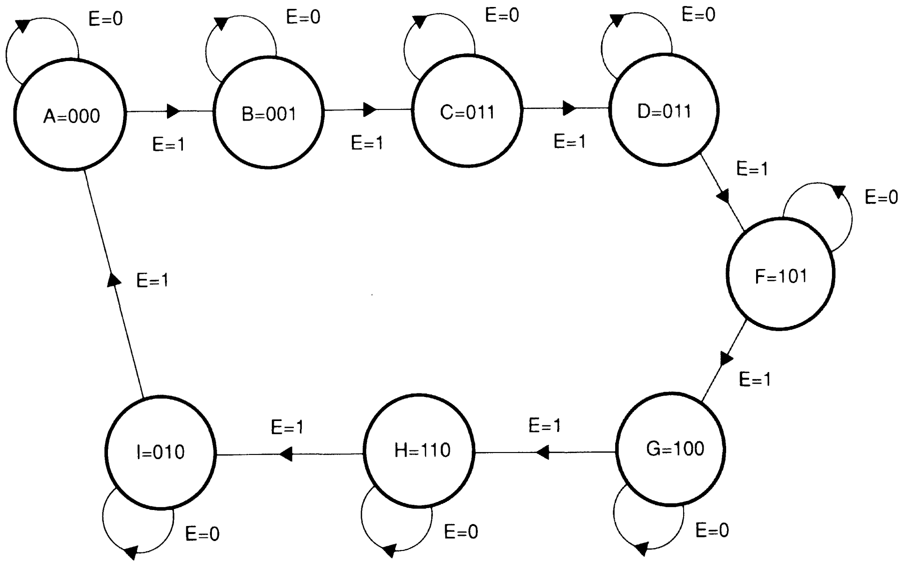
Use the proLogic compiler to create either a JEDEC file for functional verification of the design block and to program a PLD or a PDS file for translation into part of an FPGA. A batch file exists for the creation of both a `.pds` file for synthesis and a `.jedec` file for PLD programming. The design flow is shown in Figure 8-16.

Figure 8-16. Design Flow Using TI ProLogic Software



Each step in the flow reflects how smooth a process designing FPGAs with the TI design environment can be. The state machine has eight states that drive another component part of the memory arbitration unit. Figure 8-17 describes the eight states that the Moore machine must follow.

Figure 8-17. State Machine Description



8.2.7 ProLogic Design Flow

The source file in Figure 8-15 indicates that the target device is a 22V10 bipolar PLD with three inputs and three outputs. The *define* statement assigns signal names to the specified pins; e.g., *Q2* is the signal name for Pin 20.

The state diagram is self-explanatory with *if-else* constructs used to monitor the current state and the next state. Because of the nature of the 22V10 PLD, it is necessary to enable the output flip-flops of the device in the required manner using the statement *Qx.oe=1*.

Finally, a test vector block can be added as shown in Figure 8-15. The test vector block describes the inputs, outputs, and all possible conditions of the machine.

After the proLogic source file is described, the next step is to create a JEDEC file, shown in Figure 8-18, and then simulate the proLogic source file to ensure that it is functionally correct. The simulator uses the fuse-list lines from the JEDEC file to create a device model and executes the test vectors against this model, storing the results in a `.tst` file (Figure 8-19).

Figure 8-18. Example Runs of JEDEC Compilation and Simulation Run

```
*****************************************************
ProLogic Compiler
TEXAS INSTRUMENTS V2.1
Copyright (C) 1992 ProLogic Systems
Preprocessing define, include & repeat statements'
Processing statements
Executing Attributes
Minimizing logic
Creating signal specifications
Device mapping
Creating fuse plot
Creating Jedec output
ProLogic simulator
Texas Instruments V2.0
Copyright (C) 1991 ProLogic Systems
Model Compilation
Device simulation detects no errors

*****************************************************
```

Figure 8-19. The .tst File (Simulation Result File)

```
ProLogic Compiler
TEXAS INSTRUMENTS V2.1
Copyright (C) 1992 ProLogic Systems

Architecture Description: p22v10.lxa
JEDEC Fuse Information:  mog2.jed
JEDEC Test Vectors:  mog2.jed

V01  000N NNNN NNNN NNNN NNNL LLNN
V02  C0XN NNNN NNNN NNNN NNNL LLNN
V03  C10N NNNN NNNN NNNN NNNL LLNN
V04  C11N NNNN NNNN NNNN NNNL LHNN
V05  C11N NNNN NNNN NNNN NNNL HHNN
V06  C11N NNNN NNNN NNNN NNNH HHNN
V07  C11N NNNN NNNN NNNN NNNH LHNN
V08  C11N NNNN NNNN NNNN NNNH LLNN
V09  C11N NNNN NNNN NNNN NNNH HLNN
V10  C11N NNNN NNNN NNNN NNNL HLNN
V11  C11N NNNN NNNN NNNN NNNL LLNN

No errors detected with 11 Test Vectors.
```

8.2.7.1 Creating a PDS File

The next step in the flow is to create an input file for synthesis into ADL and subsequent optimization. The `.pds` file is created at the same time as the `.jedec` file but can be created separately using the **LC design-P** proLogic command. Figure 8-20 shows the resulting `.pds` file for the state machine.

Figure 8-20. The `.pds` File for Synthesis and Optimization

```
CHIP RE E nc nc nc nc nc nc nc
nc GND nc nc nc nc nc nc nc Q2
Q1 Q0 nc VCC

EQUATIONS

Q2.TRST = VCC;

Q1.TRST = VCC;

Q0.TRST = VCC;

Q2:= Q1 * Q0 * RE * E
        + Q2 * RE * /E
        + Q2 * /Q1 * RE

Q1:= Q2 * /Q0 * RE * E
        + /Q2 * Q0 * RE * E
        + Q1 * /Q2 * RE * /E
        + Q1 * Q0 * RE * /E;

Q0:= /Q2 * /Q1 * E * RE
         + Q0 * Q1 * RE
         + Q0 * /E * RE
```

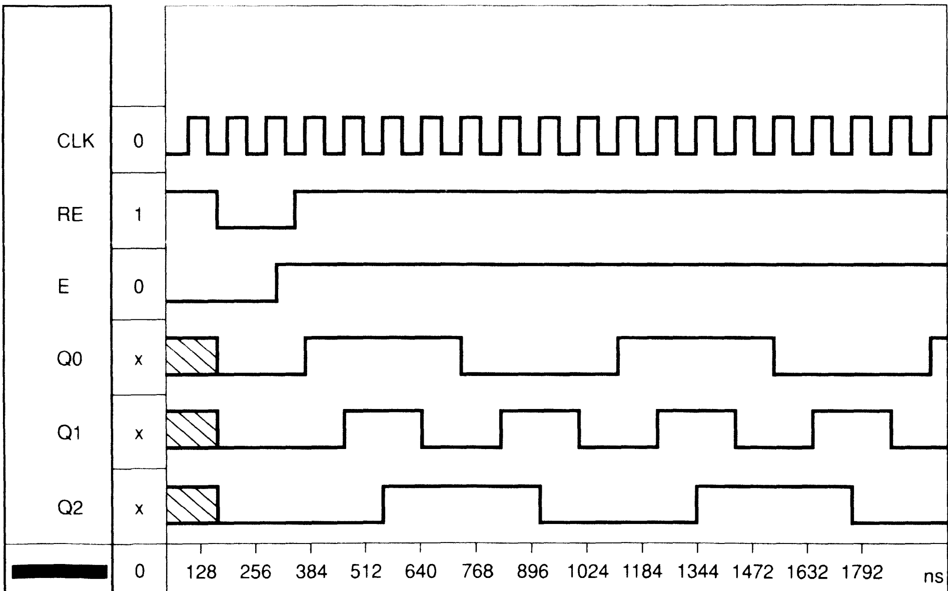
8.2.7.2 Creating ADL and a Wire File for Viewlogic

The synthesis and optimization tool converts the `.pds` file into an EDIF file that creates an ADL and wire file. In this case, the EDIFNETI utility generates a Viewlogic wire file to integrate the state machine with the complete memory arbitration schematic. To simulate the state machine in the Viewlogic environment, a symbol must be created with an underlying wire file that in this case is generated from the EDIF file produced by ALES.

The simulation of the state machine in the Viewlogic environment produced the results shown in Figure 8-21.

Viewlogic Viewgen software is a useful tool that can help investigate erroneous simulation results (Viewgen is not currently part of the design environment but demonstrates the capability of the proLogic and TI-ALS tools). This creates a schematic from an existing wire file.

Figure 8-21. Viewlogic State Machine Simulation Results



For this particular state machine the schematic is shown in Figure 8-22. As expected the state machine is implemented using three flip-flops and combinational logic. The wire file can be used to integrate the state machine block into the rest of the design for a complete FPGA design.

Figure 8-22. Viewgen-Created Schematic of Block

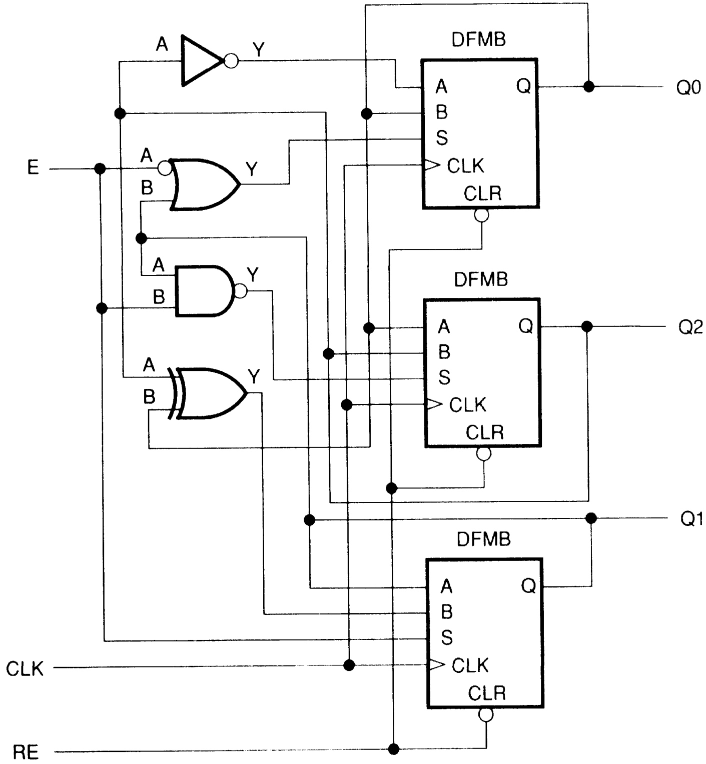
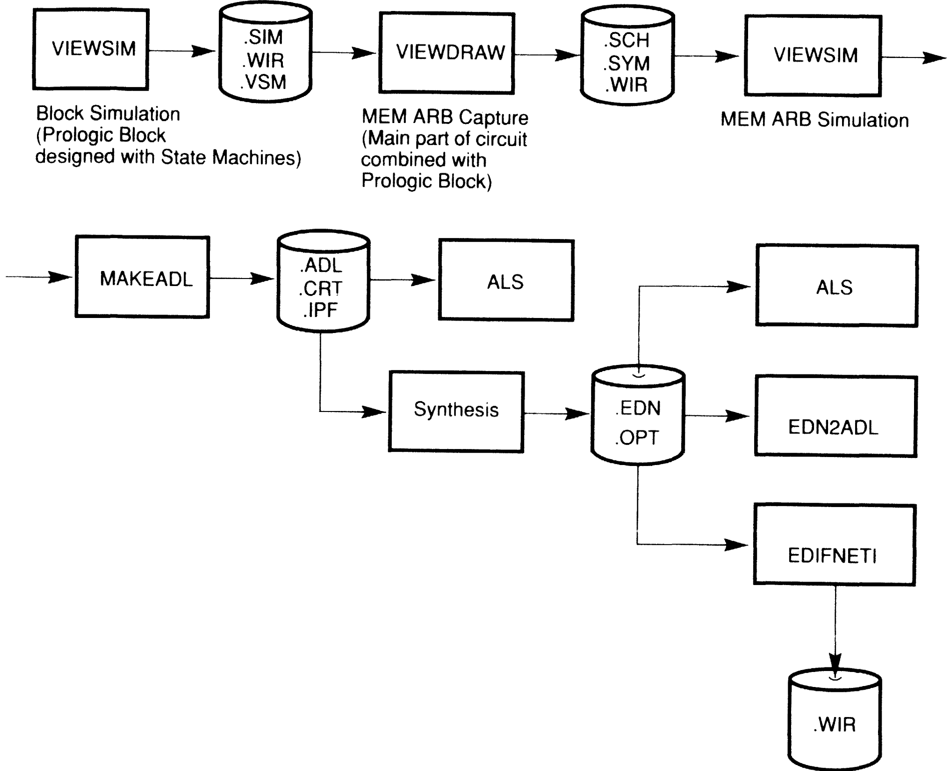


Figure 8-23 shows the flow for achieving this. If the state machine block is integrated with the rest of the systems a new complete FPGA wire file will be generated which will then be used to generate a new ADL for the whole FPGA.

Figure 8-23. Integration of proLogic Block With Schematic Blocks



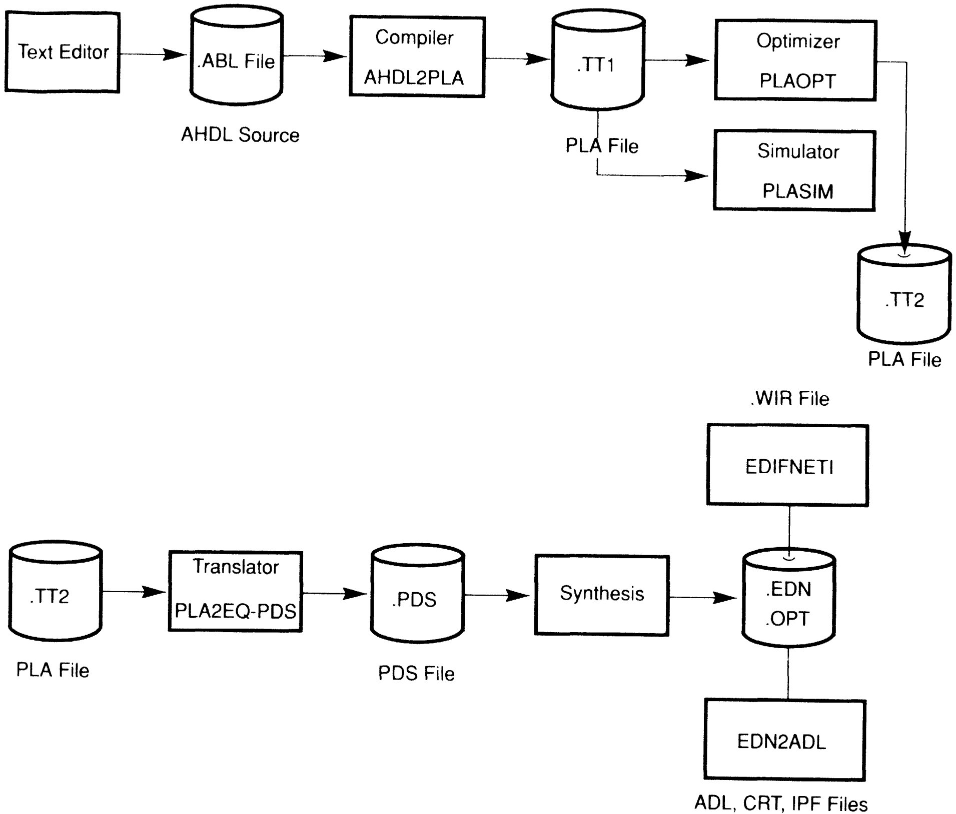
8.2.8 Design Flow Using the ABEL Design Software

The ABEL design software from Data I/O offers another approach for state machine design. The ABEL file is shown in Figure 8-24. The complete design flow is shown in Figure 8-25.

Figure 8-24. ABEL File for State Machine Design

```
module mog
          mog device 'P22V10';
"Inputs
     clk  pin1;     RE  pin2;     E  pin 3;
"Output
          Q2.Q1.Q0 pin 21.22.23 is type 'buffer.reg_D';
module mog
          mog device 'P22V10';
"State Register assignment
           sreg         = {Q2.Q1.Q0};
           A            = {0.0.0};"use one bit changes for better optimization
           B            = {0.0.1};
           C            = {0.1.1};
           D            = {1.1.1};
           F            = {1.0.1};
           G            = {1.0.0};
           H            = {1.1.0};
           I            = {0.1.0};
Equations               sreg.ar = IRE;      sreg.clk = clk
state_diagram sreg
state A: if E then B else A;
state B: if E then C else B;
state C: if E then D else C;
state D: if E then F else D;
state F: if E then G else F;
state G: if E then H else G;
state H: if E then I else H;
state I: if E then A else I;
test_vectors
          {[clk.RE.E] -> [sreg]}
           [c.0.1] -> [ A ]}
           [c.1.1] -> [ B ]}
           [c.1.1] -> [ C ]}
           [c.1.1] -> [ D ]}
           [c.1.1] -> [ F ]}
           [c.1.1] -> [ G ]}
           [c.1.1] -> [ H ]}
           [c.1.1] -> [ I ]}
           [c.1.1] -> [ A ]}
           [c.1.1] -> [ B ]}
           [c.1.1] -> [ C ]}
           [c.1.1] -> [ D ]}
           [c.1.1] -> [ F ]}
           [c.1.1] -> [ G ]}
           [c.1.1] -> [ H ]}
           [c.1.1] -> [ I ]}
 end
```

Figure 8-25. ABEL Design Flow



The AHDL2PLA compiler is used to convert the ABEL (`.abl`) file into a file in PLA format. The PLAOPT tool optimizes this file conversion into PDS. The PLA2EQN utility within ABEL generated a `.pds` file that is then entered into ALES. The ALES software generates an EDIF file that can be converted into a `.adl` file for the TI-ALS environment. The rest of the design flow is the same as that for the proLogic controller described in Section 8.2.7.

In addition to the proLogic and ABEL approaches, FPGAs can be designed using schematics; that is, designing the state machine on paper using transition maps and entering the complete schematic. Although this approach can be time-consuming, it is an option if you are designing less than 20K gates and can relate to schematics more easily than blocks of equations.

8.2.9 Design Implementation in a TPC1010A

After the design is complete and each block is simulated both individually and as a complete design to verify functionality, it is necessary to place and route the design prior to programming a device. In order to carry out the operation, the TI-ALS is used as described in Section 3.1. The I/Os of the design are automatically assigned to the device pins with the exception of the CLK pin, which is fixed to the CLK buffer pin. This is connected to the clock distribution network to minimize clock skew.

It is useful to bring out the key elements of the system functionality to demonstrate the flexibility that FPGA gives the designer. The signals generated by the controller allow the designer to refresh memory as well as control memory interaction with the rest of the system. It is important to note that while this particular DRAM controller has four row strobes and one column strobe and a column and page mode, this is just an example of the type of controller that can be effectively implemented in an FPGA. The Controller design is based on a series of state machines which can be designed in various ways. Using the design environment and by focusing on a small part of the total design we have demonstrated that the designer has the option of entering the design in a schematic form or with state machines. The ability to simulate functional blocks prior to implementation gives the designer complete confidence that the system functions correctly and this capability, together with the TI place and route and timing analyzer tools give the designer a very powerful design environment.

8.3 Implementing CPUs With FPGAs†

This section describes the implementation of a simple CPU on a single TI TPC10 series FPGA and emphasizes the issues most affected by FPGA implementation: how internal data paths are best implemented, the effect of fixing I/O pins to achieve a standard interface with other system components, how much processing capability an FPGA can hold, and the speed at which the processor can execute instructions.

This last point is focused on in detail from the standpoint of proper CPU operation, and more realistically, when it is part of a system whose overall correct behavior is of the most pertinent concern. A thorough discussion of this case will give an understanding of the issues and timing characteristics involved when interfacing an FPGA-based design with other components, such as memory devices, that provide data to or accept data from the functional units residing within the FPGA.

The OrCAD CAE environment was used in this project. Some points concerning use of this CAE package with the TI-ALS system are also discussed.

8.3.1 Overview of CPU Design

As our project progressed, increasingly complex designs were implemented. We started with a simple 4-bit CPU to learn and understand the concepts involved as well as to get a feel for the functionality the selected devices would accommodate. From this we progressed to an 8-bit CPU which we eventually maxed out with as much functionality as could be fitted into a TPC1020 part. The design overview describes the maximum functionality design (referred to as Version 3.2). Any points concerning the other designs (which are subsets of the v3.2 design) are made only in the results section if they provide relevant or enlightening information.

The CPU is essentially a complex sequential network performing assembly language instructions. It is designed to continuously fetch and execute instructions. The instructions and data are stored in an external memory; thus, much of the CPU activity involves transferring information into and out of the CPU. The remainder of the activity involves transferring information along internal data paths, one of which includes the ALU functional block which performs logical or arithmetic operations on the data. A block diagram of the v3.2 CPU is shown in Figure 8-26. Table 8-9 shows the instruction set for the CPU.

† Contributed by Mark W. Hunter and Scott R. Smith, Department of Electrical Engineering, Southern Illinois Unitversity.

Figure 8-26. CPU Block Diagram

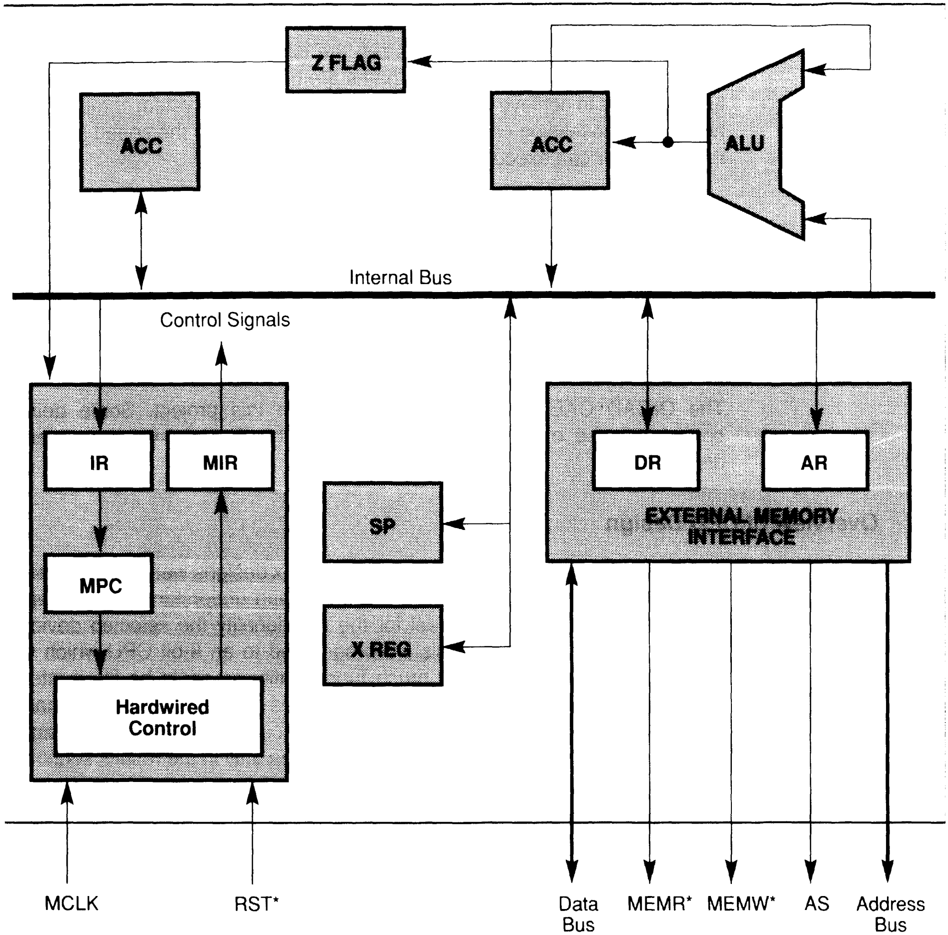


Table 8-9. List of Instructions Supported by v3.2 CPU

Opcode	Operand	Instruction	Descriptions
			Data Transfers:
3A	xx	ENTER	Enter xx in ACC
8A	xx	LOAD	Load ACC w/ data at address xx
90	xx	STORE	Store ACC at memory address xx
			ALU Operations:
10		INV	Invert contents of ACC
81	xx	OR	Logical Or ACC and data at xx
86	xx	SUBT	Subtract data at xx from ACC
89	xx	ADD	Add ACC and data at xx
8B	xx	AND	Logical And ACC and data at xx
A0		INCA	Increment ACC
			ACC Operations:
11		SHL	Logical shift ACC left by 1 bit
12		SHR	Logical shift ACC right by 1 bit
			Jump Operations:
32	xx	JUMP	Jump to location xx in program
B0	xx	JUMPZ	Jump to location xx if ACC=0
			Index Register Operations:
20		INX	Increment X register
30	xx	LDX	Load X register with xx
4A		LX	Load ACC w/ data at X address
5A		LX+	Load ACC w/ data at X address; X++
60		SX	Store ACC data at X address
70		SX+	Store ACC data at X address; X++
			Stack Operations:
31	xx	LDSP	Load stack pointer with xx
C0		PUSH	Put data on top of stack
DA		POP	Take data off top of stack
E0	xx	CALL	Call subroutine at location xx
F0		RET	Return from subroutine

An instruction is fetched and executed via a defined sequence of operations referred to as micro-instructions. Each micro-instruction is performed during one period of the CPU main clock signal and typically involves a single simple data transfer or operation. To illustrate the principal ideas of CPU behavior the sequence of nine micro-instructions that implement the ADD instruction are listed in Table 8-10.

Table 8-10. Sequence of Nine Micro-Instructions Implementing the ADD Instruction

Fetch Cycle (common to all instructions)	
AR <- PC	Transfer PC value to address register
DR <- M(AR)	Initiate read of memory data
IR <- DR; PC++	Finish reading memory data into IR; increment PC
MPC <- IR	Transfer IR value to MPC
Execute Cycle (unique for each instruction)	
AR <- PC	Transfer PC value to address register
DR <- M(AR)	Initiate read of memory data
AR <- DR; PC++	Finish reading memory data into AR; increment PC
DR <- M(AR)	Initiate read of memory data
ACC <- ACC+DR	Finish reading memory data, add it to ACC value and put result into ACC

All instructions are similarly defined by such a microprogram, differing only in the sequence and functionality of the execute cycle micro-instructions. Note that assembly level instructions are built by piecing together a relatively small number of micro-instructions. A large fraction of the assembly level instructions of any processor are data manipulation instructions which are implemented exactly as the ADD instruction with the only exception being that a different ALU function is selected during the final micro-instruction.

The ADD microprogram illustrates two additional points worth noting. First, as can be seen in two instances, more than one operation may occur during a micro-instruction. As long as the operations performed do not interfere with each other, such as both trying to drive the internal bus, multiple operations can occur.

The second point concerns the memory read cycle which is made up of three micro-instructions (for instance, the first three instructions of the ADD microprogram). The ADD instruction contains three read cycles (the last two are partially overlapped). Each read cycle is similar with the exceptions of the source of the address and the destination of the data. The read cycle is discussed in more detail in Section 8.3.2.4.

The CPU control unit (CU) produces the control signals that select and initiate each micro-instruction action. Control signals select data paths and operations and define the states of external control signals such as the memory read signal (MEMR•). The CU includes a microprogram counter which keeps track of what the CPU should be doing at any time (this counters value is influenced by the instruction code held in the IR). The MPC is the input, along with the CPUs flag(s), to a block of logic that outputs the correct control signal values for the specified micro-instruction. Control signals are latched in the micro-instruction register.

A micro-instruction is implemented over the duration of a clock period as follows. The clock's rising edge alters the microprogram counter so that it correctly specifies the next micro-instruction. This changes the inputs to the control signal logic which then produces the control signals pertaining to the new micro-instruction. The clock's falling edge then latches these into the micro-instruction register which allows the control signals to be broadcast throughout the CPU. Once the data has moved along the designated path, perhaps being operated on along the way, the clock's rising edge latches the result into the appropriate destination register (specified by a control signal) thus finishing the micro-instruction. Note that this second rising edge also initiates the next micro-instruction; thus while the contents of newly altered registers are settling, the next set of control signals are being derived.

Detailed schematics of the CPU are not included here. (Full schematics are available on the TI FPGA bulletin board service, contact TI Applications support.) To describe how detailed schematics were obtained, the next section will discuss how the CPU blocks were implemented using the macro library. In that section a detailed schematic of one control signal is also given to describe the procedure used to implement the random, wide-input control circuitry. Further specifics of actual CPU operation are given only if needed in the results section when discussing timing analyses.

8.3.2 CPU Implementation

This section discusses implementation of the CPU design using TI FPGAs. Topics include design considerations for the targeted implementation as well as those concerning use of the development tools. Because general aspects of these topics are covered extensively in other TI literature, only particular details pertaining to the CPU design are covered.

8.3.2.1 Internal Bus Considerations

Because the TPC10 series does not support internal 3-state buses, a less traditional implementation of multiple-driver data paths using multiplexers is required. The TI FPGA logic modules can implement multiplexers very efficiently allowing easy and effective implementation of our CPUs internal bus.

This approach actually proved beneficial in terms of overall chip area required to implement our CPU design. The primary reason for this is that the necessary number of control signals was decreased. Instead of a control signal for each potential driver, the driver of the internal bus was specified in an encoded manner. In our case, with five potential bus drivers, the generation of three control signals sufficed in contrast to the five required for individual enable control signals.

The multiplexer approach actually turns out to be a case of "vertical microcoding" which requires fewer control signals but needs more time to decode the control signals than "horizontal microcoding" with more controls signals that are available sooner. Further assessment of this speed/area trade-off really depends on how 3-state drivers are implemented on alternate FPGA designs. The vertical microcodings decode time is made up of the TPC10s multiplexer delay which is added to the data delay through the multiplexer. This time should be compared with the propagation delay through a 3-state buffer while also considering the time to switch from a high impedance to a driving state. Additionally, when comparing areas required by the alternatives, the area for centralized multiplexers could turn out to be less than that required to add 3-state driving capabilities to every bus driver. A final point concerns expansion of the number of bus drivers.

Here, the multiplexer approach is more granular in that the number of multiplexer inputs are powers of two. For TPC10 FPGAs going above four inputs necessitates more than one logic module thus creating a nontrivial increase in the area and propagation time.

8.3.2.2 Control Unit Considerations

Before going to the FPGA implementation of our CPU design, its control unit was of the microprogrammed variety. This approach makes use of a rather large read-only control memory which stores the control signal values for each micro-instruction. The inefficient implementation of memory with the FPGA architecture was avoided by generating a random logic circuit for each control signal (our v3.2 design had 18). The circuits had the original control memory addresses as their inputs. The amount of random logic was minimized by taking advantage of the fact that of the 512 possible address input permutations only about 60 were actually used. Thus, this approach took less area than supporting the overhead circuitry of a memory which allows every address to contain any value.

To find an optimal representation of these logic functions their truth tables were entered into a standard logic reduction routine (supplied by OrCAD's PLD program). This technique produced a minimal sum of products expression for each control signal. This expression was then implemented with several levels of logic gates to produce the hardwired design that implemented the function of nine variables.

An example of one of the control signals logic circuit illustrates this implementation. Figure 8-27 shows the circuit for control signal #1, for which the logic equation is shown in Table 8-11:

Figure 8-27. Hardware Implementation of Control Signal Latching Address Register

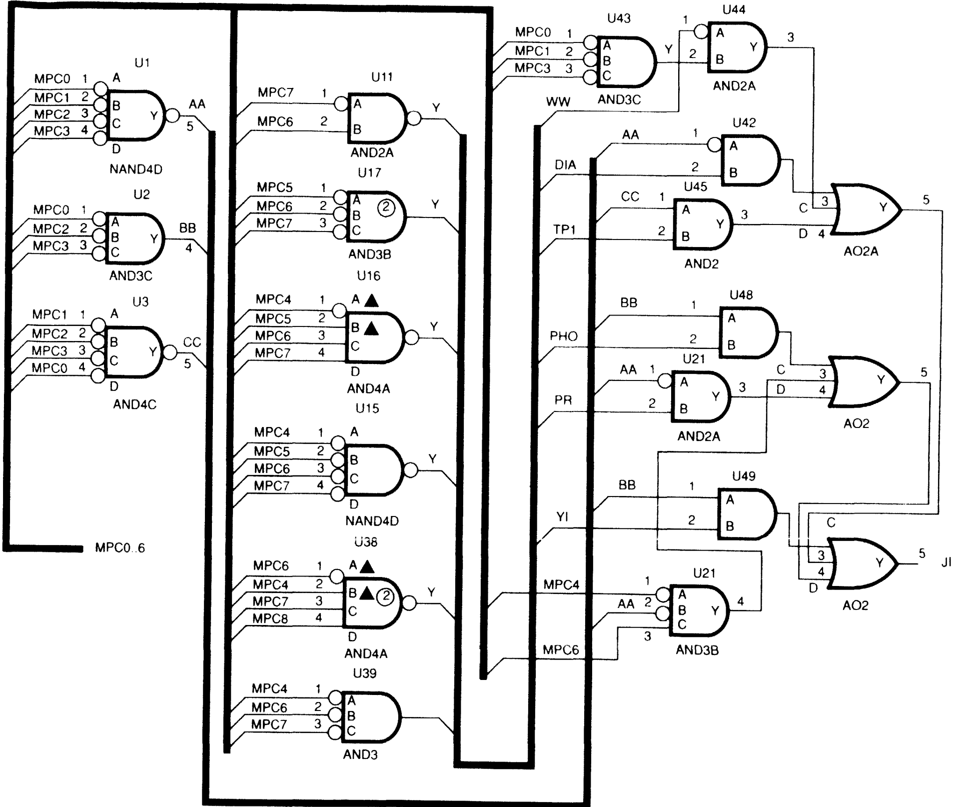


Table 8-11. Logic Equation for Control Signal #1

C1 = (MPC0&MPC1&MPC2&MPC3&MPC4&MPC6&MPC7&MPC8)
#(MPC0&MPC1&MPC3&MPC4&MPC5&MPC6&MPC7)
#(MPC0&MPC2&MPC3&MPC4&MPC5&MPC6&MPC7)
#(MPC0&MPC1&MPC2&MPC3&MPC4&MPC6&MPC7)
#(MPC0&MPC1&MPC2&MPC3&MPC4&MPC6)
#(MPC0&MPC1&MPC2&MPC3&MPC6&MPC7)
#(MPC0&MPC2&MPC3&MPC5&MPC6&MPC7)

NOTE: The symbol & signifies AND, # signifies OR, and ' signifies NOT.

It should be apparent from the example that this procedure was very tedious and not assured to be optimal for the FPGA architecture. The use of the ALES software (not supported in the OrCAD environment) would have been very beneficial in this part of the design.

The CPU block diagram in Figure 8-26 shows that our design is basically interconnected registers and counters and an ALU. These functional units were implemented using the macros TI provides with the TI-ALS (many of which were the TAxxx series of TTL look-alikes). We had to implement the ALU macro (the TA181 is no longer part of macro library TI provides, although the field applications engineers did provide a similar macro). This experience taught us that creating and modifying a soft macro was a relatively simple procedure of interconnecting existing macros and specifying behavioral attributes in a file required by the OrCAD simulator. Existing or user-developed soft macros can also be modified, in contrast to actual TTL ICs, to fit our particular needs. For example, although not implemented in our current design, an overflow flag could easily be added by making the ALU next-to-most significant carry bit available.

8.3.2.3 TI-ALS/OrCAD Interface

The OrCAD CAE software is used to enter and simulate the CPU design. As part of the TI-ALS development system, TI supplies library files containing hard and soft macros for the OrCAD/SDT schematic entry program. The OrCADs LIBEDIT facility was used to modify the ALU macro as discussed above. In this way the macros with which our design is built were easily available for interconnection using the schematic capture program. There were no special considerations required to use SDT for an FPGA design but for one point. This concerned the fact that the power sources of the design must be the V_{CC} and GND entities supplied by the TI-ALS system in the POWER library. You should not use OrCAD's standard power sources obtained using the PLACE power command. Once the design is entered, a TI-ALS compatible netlist is generated using the TI-supplied OR2ADL program.

The OrCAD environment does not support two of the optional capabilities available when using the TI-ALS software with the Viewlogic CAE environment: the use of the ALES logic enhancer/synthesizer and design simulation using actual post-routed delays.

The ALES logic enhancer/synthesizer, which allows designs to be entered as Boolean equations, interfaces only to the Viewlogic Viewdraw program. This capability would have made the design of the control unit's random control logic much easier and faster. (We did use OrCAD's PLD program to automatically synthesize minimal sum-of-products expressions from truth table specifications, but we had to manually enter the gate designs which were not actually optimized for FPGA implementation. See Figure 8-26.)

One way to benefit from ALES logic optimization for FPGAs while using the OrCAD environment is to manually copy automatically generated Viewlogic schematics into OrCAD form (for example, to obtain the ALU macro design for the OrCAD environment).

The ALES package can also optimize a design for TPC10 FPGA implementation according to speed and/or area criteria. Since we were unable to take advantage of this, it may be the case that the functionality or speed of our final CPU design may have been slightly inferior to a design implemented in conjunction with the automatic optimization tool.

OrCAD's VST simulation package does allow functional simulation (using the TI-supplied simulation models) which is a very useful tool for design verification. But actual delays from the TI-ALS place and route tools cannot be back annotated to the simulator to show simulation results. This timing information can be extracted using the TI-ALS timer utility, although obtaining the exact path delays required is not always straightforward, as will be discussed below.

These considerations seem to warrant switching CAE environments to Viewlogic if one is going to do more than a trivial amount of FPGA design.

8.3.2.4 TI-ALS Environment

The TI-ALS environment accepts a design netlist and provides the tools to do automatic place and route, timing analyses, and actual programming of the FPGAs.

One of the facilities of the TI-ALS system allows fixing I/O signals to particular external pins. We took advantage of this for our CPU designs so that every CPU we implemented had the same interface to the RESET signal and to external memory and I/O devices. Note that the power and clock signals are fixed in any case. We were somewhat concerned about the ability to successfully route a design with a large module utilization percentage when fixing all pins, but this was never a big problem for the small number of I/O signals involved in our designs (20). The trade-off for fixing pins is not allowing the TI-ALS system to optimize minimum delays and maximum utilization as it attempts to during automatic pin assignment.

The Validate function of TI-ALS does a design rules check of a design versus compatibility with the device to be programmed. A common error/warning encountered in our designs dealt with fan-out limitations. Nets (with default criticalities) produce a warning if their fan-out exceeds 10 and an error if fanout exceeds 24. Such errors and warning were corrected by placing buffers in the design. Note that many soft macros hide the number of inputs actually presented to an incoming signal; in fact, sometimes the internal circuitry of a soft macro will present enough inputs to cause a warning message.

The TI-ALS combiner function will remove logic modules from soft macros that are determined to be unneeded for correct functionality, thus performing some automatic optimization to reduce module count. This decreased the actual number of modules used in our v3.2 CPU design by 30. After the combiner is executed a final tally of logic modules and errors and warnings is given as well as statistics on nets and their fan-outs. From this a probability of successfully routing the given design is reported.

The configure, automatic place and route, create antifuse file, and activate facilities are then run to actually create a programmed FPGA. TI-ALS allows the user to define critical signal paths (to be made as fast as possible) which the place and route routines give higher priority than other nets. We programmed our CPU onto 68-pin TPC1010 or TPC1020 devices, although the 44-pin packages would have sufficed for our purposes. These facilities create `.loc` and `.del` files which allow the user to see where certain macros are placed and the delays of certain nets, respectively.

The timer facility of TI-ALS can be used after placing and routing a design to check out the delays along certain paths in a design. We used this function to characterize the speed at which we can run our CPU. The timer allows the user to define start and end sets which specify the paths whose delays will be reported. The breakdown of these total delays into component-by-component delays (including connecting nets) is provided when using the expand command. When characterizing our design, we found that it was not always possible to define a start and end set that would cause the timer to give the exact path we were interested in because we were going off and on the FPGA chip. We also had to be careful to specify start and end sets that contained only paths actually used while the CPU was operating. The approach we adopted, then, was to think through the delay paths we were interested in and then to get the timer to give the delays along that path by specifying one or more start and end sets. The times were then accumulated by hand to give the desired total path delays.

An example of using the timer to find delays along a certain (sub) path is now shown. The path is part of the one that incoming memory data take on their way to an internal CPU register (commonly the accumulator).

The delay of the path through the data register and ALU to the accumulator inputs (see Figure 8-26) was obtained by specifying the clock to the data register control logic as the start set and the accumulator inputs as the end set. The 126.9 ns delay is expanded to show each component's contribution (see Table 8-12).

Table 8-12. Component Delay Through Data Register and ALU

Total	Delay	Typ	Load	Macro	Start Pin	Net Name
126.9	7.1	Tpd	3	OA2A	U86/U35:B	N00064
119.8	7.8	Tpd	1	AOI1A	U86/U28:B	U86/N00018
112.0	6.5	Tpd	1	A02A	U86/U19:B	U86/N00015
105.5	6.2	Tpd	1	OA5	U86/U11:B	U86/NOOO27
99.3	6.4	Tpd	1	INV	U86/U15:A	U86/N00028
92.9	10.1	Tpd	5	AO2A	U86/U37:B	N00029
82.8	8.8	Tpd	2	OA5	U85/U34:D	U85/N00011
74.0	11.9	Tpd	2	NOR2	U85/U6:B	U85/N00016
62.1	10.3	Tpd	6	AO4A	U85/U13:A	U85/N00021
51.8	30.4	Tpd	9	MX4	U54:D2	IB1_1
21.4	6.5	Tpd	2	DL1	U34:G	DR1_1
14.9	14.9	Tcq	10	DFPC	U32:CLK	NOOO04
0.0	0.0	Nsk	27		U58:Y	NOOO53

From this we can see that data (bit one) takes 6.5 ns to get through the data register latch (U34, plus the net to the next component) and 30.4 ns to get through the multiplexer (U54) which determines who drives the internal bus and to travel along the bus to the ALU inputs. The list of times regarding chips U85 and U86 are the signal propagation through the ALU soft macros and include the ripple carry connecting the two 4-bit parts. The initial 14.9 ns delay through the U32 flip-flop is from the control circuitry (generating the external read signal and opening the correct internal path through the DR) and not part of the data path we are characterizing.

The results of implementing the CPU design with TPC1010 and 1020 devices emphasizes two main aspects. First, a summary of the module requirements for the different complexity CPUs is given. This includes a discussion of how increasing the data path width affects total module count, so that readers can extrapolate our results to various widths. Secondly, a detailed timing analysis is reported for the v3.2 CPU.

The following is a summary of design implementations.

- Version 1.1 (4-bit)

  The 4-bit CPU data processing unit (DPU) used approximately 108 logic modules representing 36.6 percent of the TPC1010 device's total logic modules (295 logic modules). The DPU was composed of an ALU, accumulator, program counter, data register, address register, and instruction register.

A five instruction control unit composed of a microprogram counter and twenty-four hardwired control signals was implemented for the 4-bit design in 171 logic modules. The instruction set was not very robust, but was sufficient for the first step in the design process to produce a functional device that students could implement and expand. The total design added up to 279 logic modules producing a module utilization of 94.6 percent for the TPC1010 device.

A 68-pin PLCC package device was used for programming of this design although a 44-pin package would accommodate all of the designs. The design contained thirteen I/O pins that were all FIXED so that a generic test bed could be used. The device was then programmed on the TI Activator 1 in conjunction with a 486 PC.

- Version 2.1 (8-bit)

  Increasing the DPU from 4-bits to 8-bits increased the logic module count to 215 modules. The 8-bit design with the same CU as Version 1.1 was implemented in a total of 386 logic modules producing a 70.6 percent module utilization for the TPC1020 device (547 logic modules). There were twenty-one I/O pins for this design, that again were all FIXED for placement and routing of the design in a 68-pin PLCC package device.

  With the benchmarks provided by the 4-bit and 8-bit designs one can estimate the number of logic modules that will be required to implement an n-bit CPU of similar functionality using the equation below:

  Total # Logic Modules(LMs) = CU # LMs + [C   (DPU # LMs)]

  where

  DPU # LMs = 108 (the # for 4-bit DPU)
  C = the data-path width divided by 4

  For example:

  Total # Logic Modules for 8-bit = 171LMs + [2 • 108LMs] = 387
  Total # Logic Modules for 16-bit = 171LMs + [4 • 108LMs] = 603
  Total # Logic Modules for 32-bit = 171LMs + [8 • 108LMs] = 1035

- Version 3.1 (8-bit)

  A new DPU was constructed for v3.1 that also included a X register and stack pointer. This DPU took 347 logic modules for implementation. A CU to implement 12 new instructions for the X register and stack pointer was implemented in 200 logic modules. This gave our design a total module count of 547 representing 100-percent module utilization for the TPC1020. This design also contained 21 I/O pins. Placement and routing did fail with all twenty-one I/O pins FIXED, but was successful with just one I/O pin UNFIXED.

- Version 3.2 (8-bit)

  The final version of the SEP was similar to Version 3.1 but had an enhanced instruction set made possible by optimizing the microprograms and control unit so that more functionality could be squeezed onto a 1020 part. It was implemented in 537 logic modules, 339 for the similar DPU and 198 for a CU to implement the instruction set listed in Table 1. The module utilization for this design was 98.2%. Again, one I/O pin of the total 21 I/O pins needed to be UNFIXED to produce a successful Placement and Routing sequence. To extrapolate various data path widths for a CPU with this functionality, the reader can use the previously defined equation with (CU # LMs) as 198 LMs and (DPU # LMs) as 170 LMs.

  We performed extensive timing analyses to characterize the speed of the v3.2 CPU. This included use of the TI-ALS timer utility and timing measurements made while the chip was actually operating within a system. Two different cases were considered: in the first, the internal operation of the CPU was considered, in the second, CPU operation in conjunction with memory (and its associated delays) was characterized. Either case may turn out to be the real constraint for maximum operating frequency. Refer to the CPU block diagram for the following discussions.

  The maximum allowable internal CPU speed was determined by finding out how long the clock signal must remain high and how long it should remain low. The clock high time is constrained by the time for the control unit to do its job of generating the next micro-instructions control signals. This is made up of the time for the MPC to change (due to the clock's rising edge), for the random control logic (like that of Figure 8-27) to derive the new control signals, and for those signals to meet the MIR latch setup times (the MIR is clocked on the falling edge), plus all net traversal times. When measuring this edge-to-edge time, one must note that, in general the edges of the actual clock signal may not be directly driving the internal parts. Often the clock signal is gated and the time delay through the gate must be considered. For this particular case, the clock signal does directly drive the MPC, but an inverted clock signal drives the MIR. Thus the delay time through the inverter can be subtracted from the above path since the data need to meet the setup time before the inverted signals edge, not the earlier clock signal edge. For the particular placement and routing of our design, control signal C2 had the longest delay through the control logic and limited the high phase of the CPU clock to no less than 53.6 ns (89.3 ns [MPC Tcq of 18.1 ns plus 65.7 ns through the control logic plus 5.5 ns for MIR Tsu] minus the 35.7 ns delay of the clock inverter).

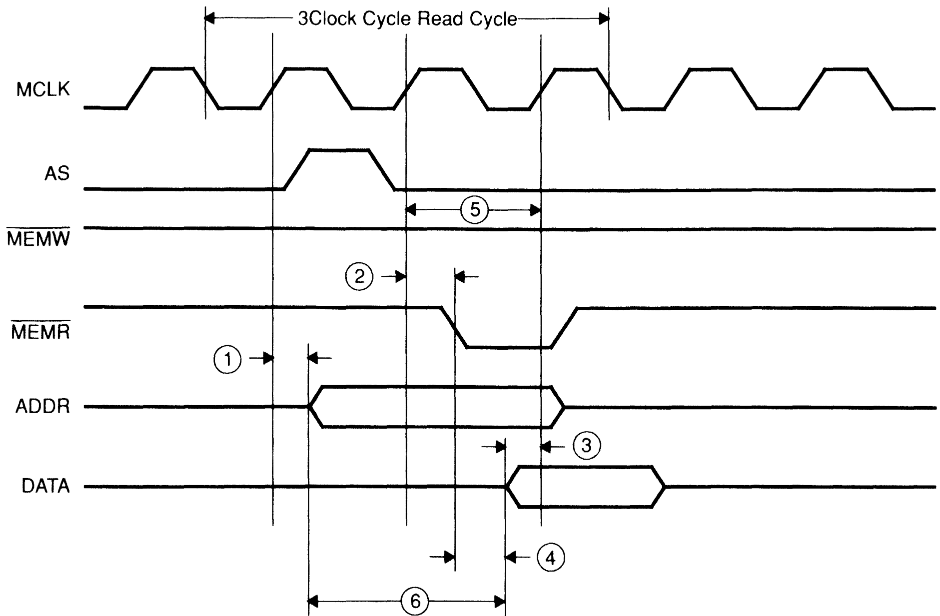
The duration of the clock's low phase is internally limited by the longest delay path that data takes as it is transferred within the chip. By considering all such transfers, we found that the micro-instruction containing the longest path is the one in which the X Register is copied into the Address Register. The low phase of the clocks is a function of the time it takes to get the control signals to the correct places (82.3 ns made up of the above-mentioned 35.7 ns clock inversion plus 46.6 ns MIR Tcq and net delays) plus the propagation delay through any data paths the control signals govern (in this case 46.0 ns through the IB multiplexer) plus the setup time for data at the Address Register (5.4 ns) minus the propagation delay of the gated clock signal to the Address Register (14.4 ns). Therefore, the low phase of the clock is constrained to be at least 119.3 ns.

For our complete system design it was determined that the worst-case delay for external data transfers occurred during a memory read cycle. Our simple system design included one memory chip which was always enabled; there was no intermediate chip-select circuitry commonly found in more complex designs. The output enable of the memory IC was connected directly to the CPU MEMR* signal. For this system setup, the time for memory to deliver data once its output was enabled was part of the worst-case delay path that restricted the clock's rising edge to rising edge time.

The read cycle delay path of concern consists of three things: 1) the time to get the MEMR* signal (triggered by rising clock edge) to the memory, 2) the time for memory to release valid data, and 3) the time it takes data to reach its destination register (which is clocked by the next rising clock edge) within the chip. A timing diagram illustrating this is depicted in Figure 8-28. Note that the actual rising edge to rising edge time constraint is a function of the speed of the memory device used in conjunction with the FPGA implemented processor. This time constraint is characterized by the equation listed at number five in Figure 8-28.

The timer tool showed the time to get the MEMR* signal to the memory to be 26.6 ns. The longest path for read cycle data once it is produced by memory is to the Z flag latch. More specifically, the data must enter the chip through a BIBUF; pass through the data register, pass through the IB multiplexer circuitry; propagate through the ALU including the ripple carry connecting the two 4-bit parts to be possibly operated on; then go to the Z flag circuitry and meet the setup time for the Z flag latch. The propagation delay for this path is 137.5 ns plus a setup time of 5.2 ns.

Figure 8-28. Timing Diagram for Read Cycle



No.	Characteristic	Min	Max	Unit
1	Clock high to address valid	-	32.3	ns
2	Clock high to MEMR* low	-	26.6	ns
3	Setup time of valid data before clock high	129.4	-	ns
4	Memory output enable time (typically Toe)	(see memory specs)	ns	
5	Rising edge to rising edge clock = 2 + 3 + 4	156.0	-	ns
6	Address valid to data valid (typically Tacc)	(see memory specs)	ns	

The latching signal for the Z flag is produced by a GAND (driven by the clock signal) in 13.3 ns. Thus, the time it takes the memory's data to reach its destination within the chip is 129.4 ns (137.5+5.2-13.3). The memory device used was a 2732 EPROM. The enable time for this device is characterized by a maximum of 120 ns. Therefore the rising edge to rising edge time for the clock is constrained to be at least 276 ns (26.6+129.4=156 ns within the FPGA plus the time it takes memory to release valid data,120 ns).

The sequences that we have defined as critical leave us with two constraints, the time to perform internal operations and the time to access external memory. Internal operations can be performed with a clock that has a high edge for 53.6 ns and a low edge for 119.3 ns. This translates into a maximum

operating frequency of 5.8 MHz. External memory can be accessed with a delay of 276 ns between rising edges of the clock or a maximum operating frequency of 3.6 MHz with no duty cycle constraints. This frequency thus defines the speed of the SEP, but one must keep in mind it is a function of the speed of memory.

When driving the actual processor with a real clock, we found its maximum operating frequency to be 5.7 MHz which corresponds to 176 ns between rising edges. A large portion of the difference between the timer analysis (276 ns) and actual delay (176 ns) was accounted for by the enable time of the memory device. The enable time observed on a logic analyzer was between 40 and 50 ns instead of the120 ns specification used to calculate the expected operating speed. This reduces our estimate to approximately 206 ns between rising edges. Therefore, we observe a difference of 20 to 30 ns between our experimental and actual delay times. This corresponds to a conservative estimate of approximately 15% for a long delay path of approximately 20 macros. A further analysis was done on a much shorter path, the delay from the rising edge of the clock signal to the time the MEMR* signal was seen by memory (see Figure 8-28, no. 2). This delay was observed to be between 24 and 28ns. This value proved very close to the timer analysis value of 26.6ns.

The test results described above show that the timer analysis supplied by the TI-ALS software is very reasonable and beneficial. A designer can most likely count on the calculated delays to be padded with a conservative estimate. We found the average delay through a level of logic to be around 8.3 ns - an interesting fact to keep in mind when designing for a speed critical design.

TI TPC 1020 FPGAs allow implementation of an 8-bit CPU of nontrivial complexity. We have described a CPU design with an instruction set of 24 instructions. Further optimization and design tricks may allow additional instructions, especially conditional jump instructions that need little more than a multiplexer for the condition to be tested (sign, overflow, and carry flags along with the implemented zero flag). Use of the TPC12 series FPGAs allows implementation of more complex CPUs and larger data buses.

Emphasis on characterizing the speed of the CPU design example illustrates the timing constraints involved when interfacing the design with external components, such as memory. Typically two parameters of a memory IC should be considered: the output enable time and the access time from presentation of a valid address and/or active chip-select signal. As shown earlier in this section, the output enable time gives the following constraint:

$T > 156 + Toe$

where:

T = the period of the main clock signal in nanoseconds
156 = the time for the MEMR* signal to be generated plus the setup time of valid data from memory
Toe = the output enable time of the memory

The access time from presentation, which is not a limiting factor in this system design, give the following additional constraint on T:

$2T > 161.7 + Tacc + Til$

where:

161.7 = the time to valid address (see 1, Figure 8-28) plus the setup time of valid data from memory

Tacc = the access time of the memory

Til = the delay through intermediate logic between the CPU and memory, such as an address decoder

The CPU design may be extended to incorporate a ready-input signal that forces wait states into read/write cycles. In this case, T times the number of wait states should be added to the left side of each equation. The timing analyses are thus general in that they show how any design incorporating similar read/write cycles can be made part of an entire system of varying complexity.

8.4 Universal Asynchronous Receiver-Transmitter (UART)†

The design example in this section describes the implementation of a UART circuit in a TI-FPGA. The signal pins used in the design are described along with the transmitter and receiver. Also presented are the structure of the simulation command file and the module requirements as specified by the TI-ALS validation software.

Although this circuit is configured to use the common configuration of 1-stop bit, 1-start bit, and no parity bit, this circuit presents a starting point for other possible transmission sequences you may desire to use.

8.4.1 Signal Definitions

The top-level circuit diagram contains a symbol that shows all of the I/O pins for the device (Figure 8-29).

Each I/O pin connects to an input or output buffer. The I/O buffers are a requirement for proper processing through the TI-ALS system and can be called up from the library under the names *INBUF* and *OUTBUF*. The I/O signals are defined as follows.

- CLK16X (INPUT)

  The master clock for the device. The clock signal on this pin is 16 times the frequency of the data transmit and receive streams.

- RESETIN (INPUT)

  Master reset for the entire chip.

- CS (INPUT)

  The active-low chip select pin, which can be used as an address pin if desired and enables the read and write pins.

- RD (INPUT)

  The active-low read pin. Once a serial data stream is received, the pin tells the receiver section that the host has read the data. It also resets the receiver section so that it can receive another serial data byte.

- WR (INPUT)

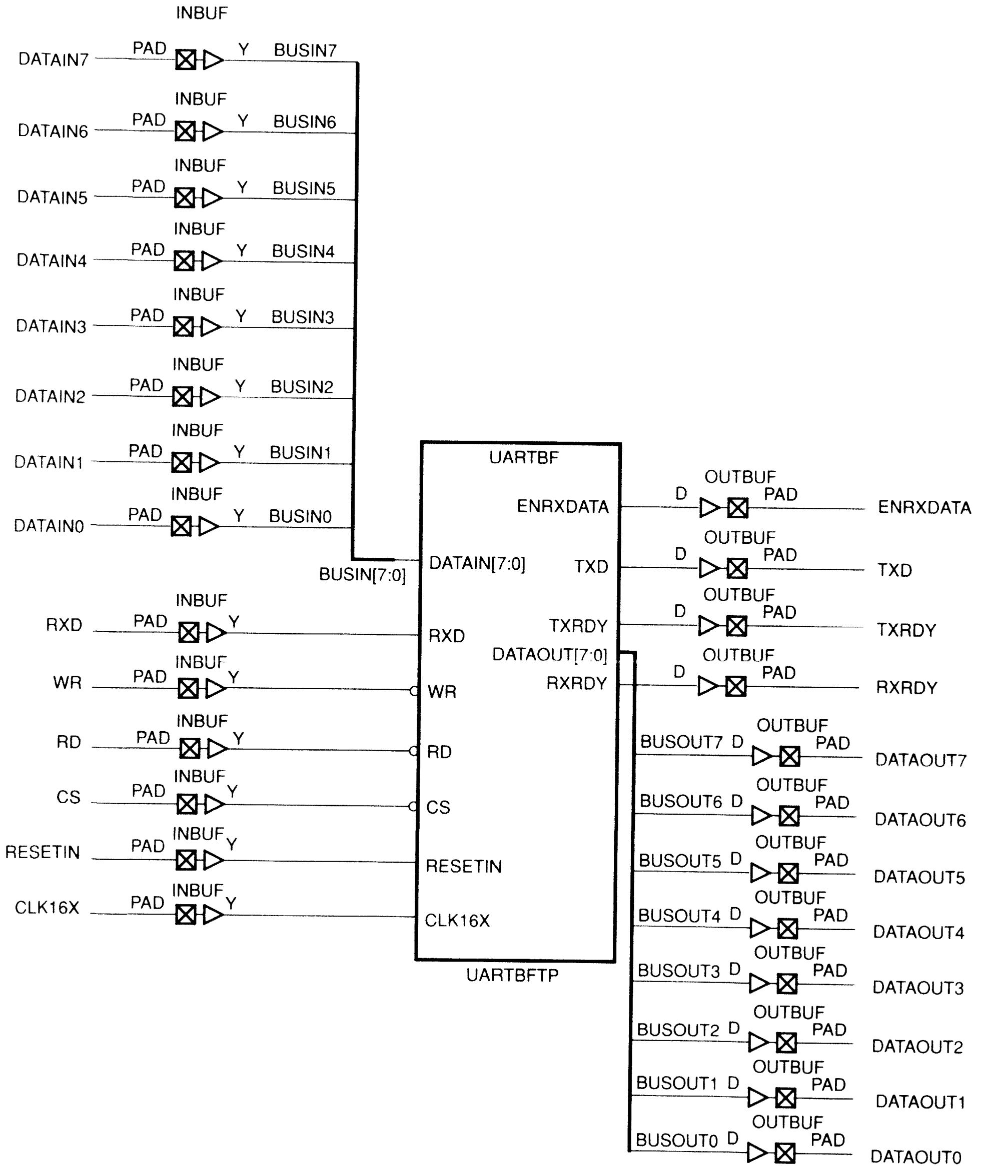
  The active-low write pin. Once data is present on *DATAIN* pins, asserting this signal latches the data and starts the transmit sequence.

- RXD (INPUT)

  Serial data receiver pin and input for the serial data stream.

† Contributed by Joel S. Lason, P. E., FPGA Applications, Texas Instruments Incorporated.

Figure 8-29. Top-Level Schematic



- DATAIN[7:0](INPUT):

  Parallel data input for the transmitter circuit; this data is registered and shifted out serially.

- ENRXDATA (OUTPUT)

  Enabled to receive data, this signal follows the read signal, indicating to an external device that the UART is ready to receive a data byte.

- TXD (OUTPUT)

  Transmit data pin. This is the serial data stream from the transmitter section.

- TXRDY (OUTPUT)

  Transmit ready. When this pin is high, the UART is ready to receive a parallel data byte for transmission.

- DATAOUT[7:0](OUTPUT)

  Parallel data output from the receiver circuit; received serially.

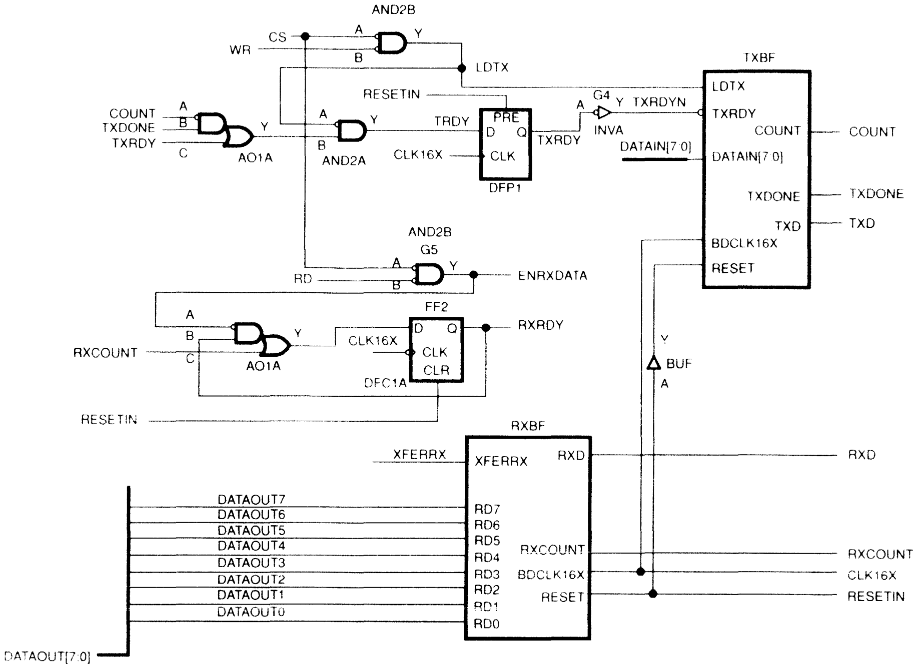
- RXRDY (OUTPUT)

  Receiver ready; this signal transitions high when a serial data stream is received and is ready to be read on the DATAOUT pins.

8.4.2 Transmitter Circuitry

Although the receiver and transmitter can operate independently of each other, they share common signals and are thus brought together one level down in the hierarchy of the schematic (Figure 8-30).

Figure 8-30. Receiver and Transmitter Sections With Arbitration



The transmitter and receiver blocks are labelled TXBF and RXBF. The additional circuits are state machines used for device status and processor interfacing.

The flip-flop that feeds the TXBF block is a state machine that indicates the status of the transmitter section. When this bit is high, the transmitter is ready to receive a data byte and begin transmission. During transmission this bit transitions to the low state to indicate that the system is not ready to receive another byte. The output signal from this flip-flop, TXRDY, is also an output from the device.

You may wish to refer to the timing diagrams shown in Figure 8-31 through Figure 8-33 during the discussion in this section.

Figure 8-31. UART Transmit and Receive Cycles

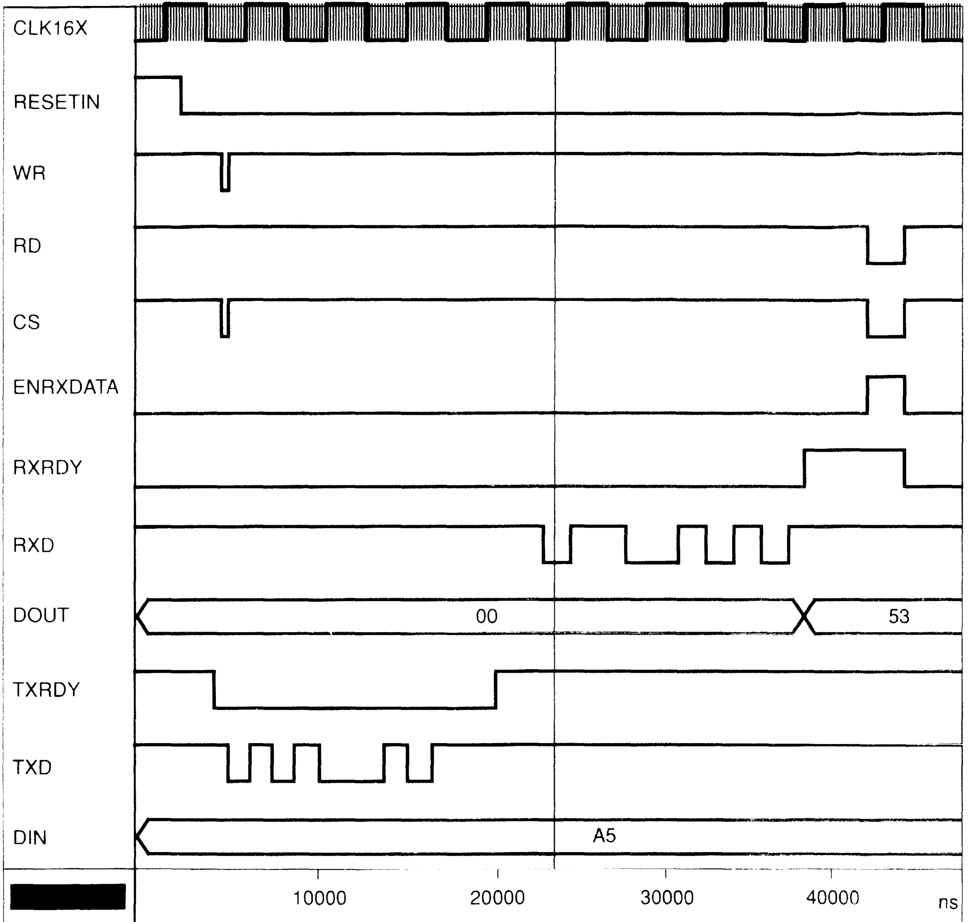


Figure 8-32. Receiver Circuit Simulation

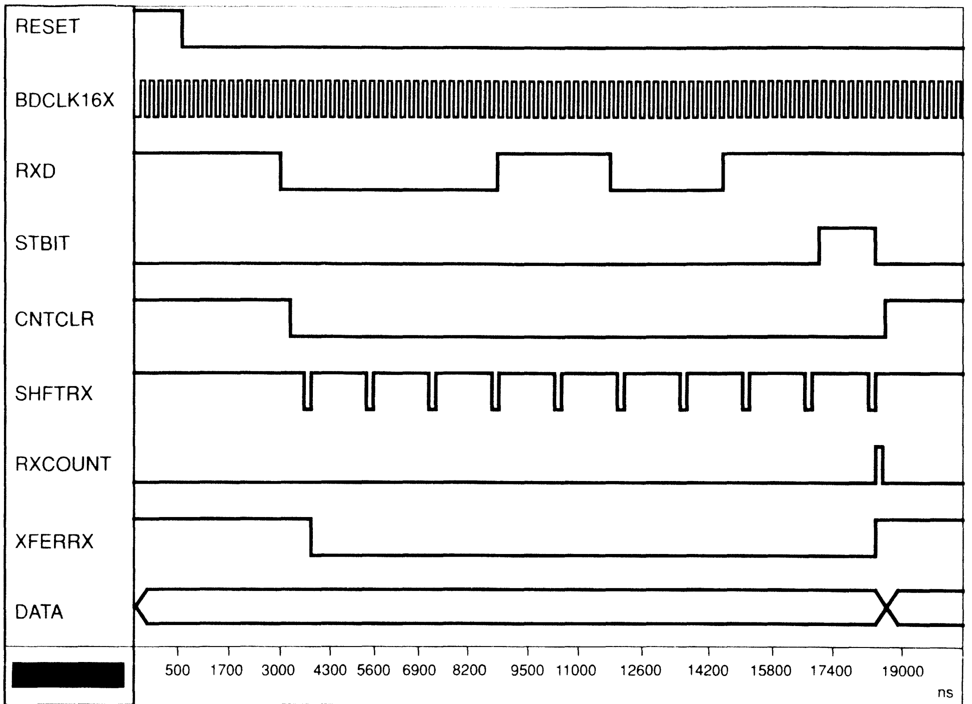
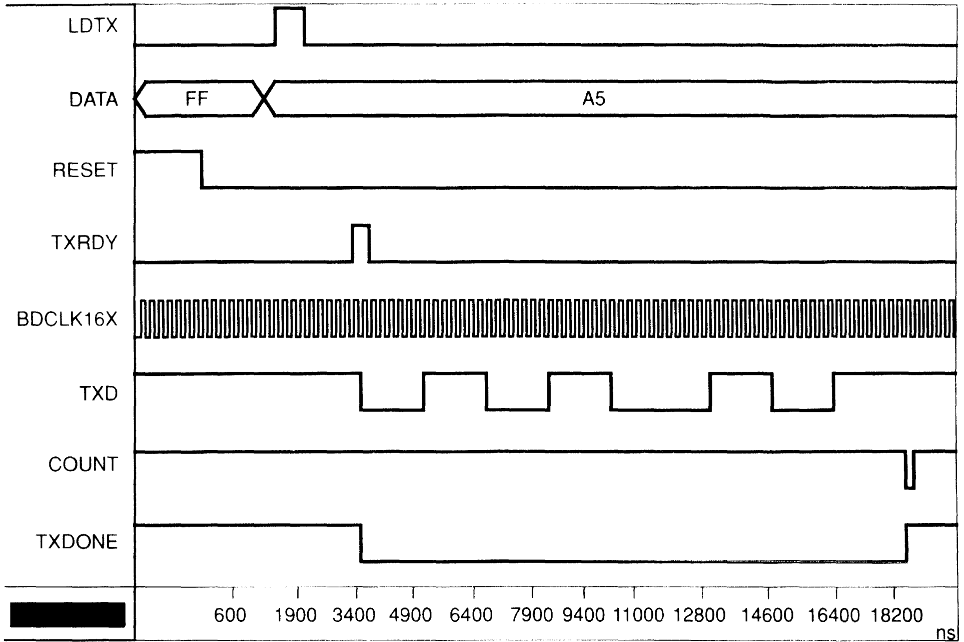
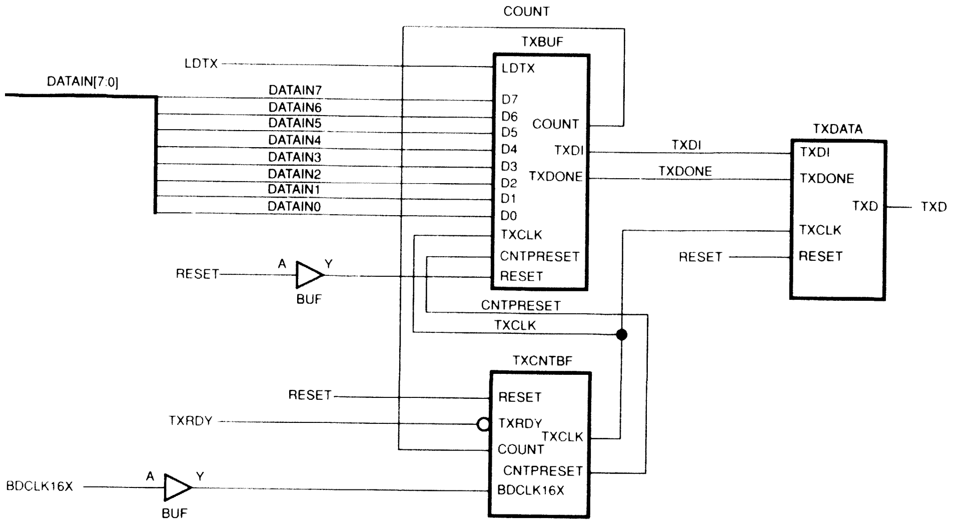


Figure 8-33. Transmitter Simulation



The subcircuits that make up the transmitter block are shown in Figure 8-34.

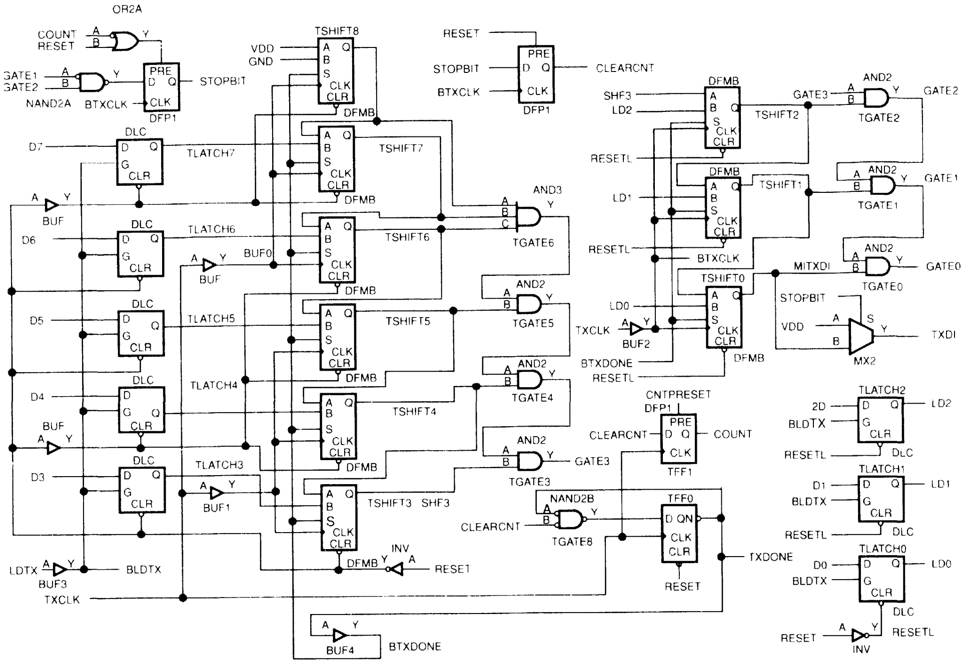
Figure 8-34. Transmitter Circuit Blocks



The transmitter block contains the transmit buffer (TXBUF), transmit controller (TXCNTBF), and serial transmit register (TXDATA) subcircuits.

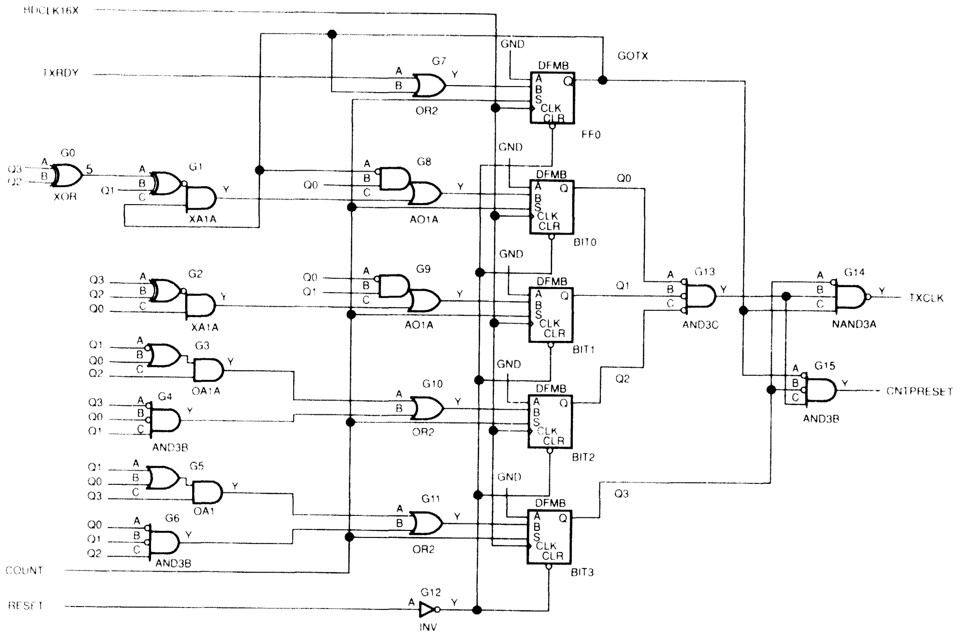
The LDTX signal tells the transmit buffer in Figure 8-35 to latch the data on the DATAIN[7:0] lines and the TXRDY state machine to go to state 0 as an indication that the device is transmitting and unable to receive another data byte (Figure 8-35).

Figure 8-35. Transmit Buffer (TXBUF)



TXRDY tells the transmit controller to begin its transmit sequence (Figure 8-36).

Figure 8-36. Transmit Controller (TXCNTBF)



The TXDONE and COUNT signals are output from the transmit buffer and fed to the TXRDY state machine to allow it to transition back into the transmit ready state, or state 1. TXD is the serial transmit pin that sources the serial output data stream.

The details of the transmit buffer in Figure 8-35 show how the data is latched and then shifted out under the control of TXDONE signal and the TXCLK generated by the transmit controller. The stopbit is also generated in this circuit.

The serial transmit register gates the transmit output stream, which is also under the control of TXDONE and TXCLK (Figure 8-37).

Figure 8-37. Serial Transmit Register

8.4.3 Receiver Circuitry

The receiver consists of two circuit blocks: the receive buffer and receive controller (see Figure 8-38 and Figure 8-39).

Figure 8-38. Receiver Circuit Blocks

Figure 8-39. Receiver Buffer

The serial input data stream is received on the RXD pin and is evaluated by the receive controller for a valid start bit. Appropriate control signals are then generated by the receive controller to sample the incoming bit in the middle of its duration. The duration of the bit should be16 cycles of the CLK16X input pin.

RXCOUNT in the receive buffer circuit tells the RXRDY state machine that the receiver has valid data that can be read and registers the shifted-in data into the second level of buffering.

The XFERRX signal is used by the receive buffer to enable the transfer of the serial input data from register to register. When the stopbit is detected, the XFERRX signal disables further shifting. SHFTRX in the receive controller circuit clocks in serial input data in the middle of its duration.

Figure 8-30 shows how the receiver status (RXRDY) state machine or state machine operates. It is initially cleared to a value of 0 to indicate that no data has been registered. RXCOUNT becomes active when a byte has been received and this sets the state machine to the 1 state.

When the CS and RD lines are made active the state machine is reset to the zero state and the receiver is ready to read more data. The RXRDY signal does not prevent further reception of data. It only signals to the processor that valid data can now be read from the device. An external device can also look at RXRDY to determine if it is appropriate to transmit another byte.

The final full simulation is shown in Figure 8-31; the simulation command file for Viewlogic Viewsim program is shown in Figure 8-40.

Figure 8-40. Simulation Command File

```
VECTOR DIN DATAIN [7:0]
VECTOR DOUT DATAOUT [7:0]
RADIX HEX DIN
RADIX HEX OUT
WAVE UARTBFTP.WFM CLK16X RESETIN WR RD CS ENRXDATA RXRDY RXD DOUT TXRDY TDD DIN
WATCH CLK 16X RESETIN WR RD CS ENRXDATA RXRDY RXD DOUT TXRDY TXD DIN
ASSIGN DIN A5\H
CLOCK CLK16X 0 1
STEPSIZE 500
H RESETIN WR RD CS RXD
CYCLE 16
L RESETIN
CYCLE 16
CYCLE 4
L CS WR
CYCLE 2
H CS WR
CYCLE 160
CYCLE 32
L RXD
CYCLE 16
H RXD
CYCLE 32
L RXD
CYCLE 32
H RXD
CYCLE 16
L RXD
CYCLE 16
H RXD
CYCLE 16
L RXD
CYCLE 16
H RXD
CYCLE 16
CYCLE 32
L CS RD
CYCLE 16
H CS RD
CYCLE 32
```

The simulation verifies transmit and receive cycles operating independently. Sublevel simulations of both the receiver and are shown in Figure 8-32 and Figure 8-33 and provide further insight into the operation of the control signals for both blocks.

This circuit, as a basic building block, can be customized for many applications as a direct result of programmability. Examples of customization are on-chip data buffer and addressing circuitry, either for address decode or output data address generation. There is plenty of room in even the smallest TI-FPGA for enhancements, because the circuit used only 40 sequential modules, 169 combinational modules, and 26 I/O modules in a TPC12 series device. The circuit fits easily in a TPC10 series device. For a copy of this database, call FPGA applications at (214)997-5666.

8.5 IEEE 1149.1 Boundary Scan Architecture†

The IEEE 1149.1 boundary scan standard was developed to offset the loss of test access brought about by the miniaturization of circuits and substrates. This standard defines a boundary scan test structure and 4-pin test bus interface that can be designed into ICs to provide a method of testing the wiring interconnects between ICs assembled on a common substrate.

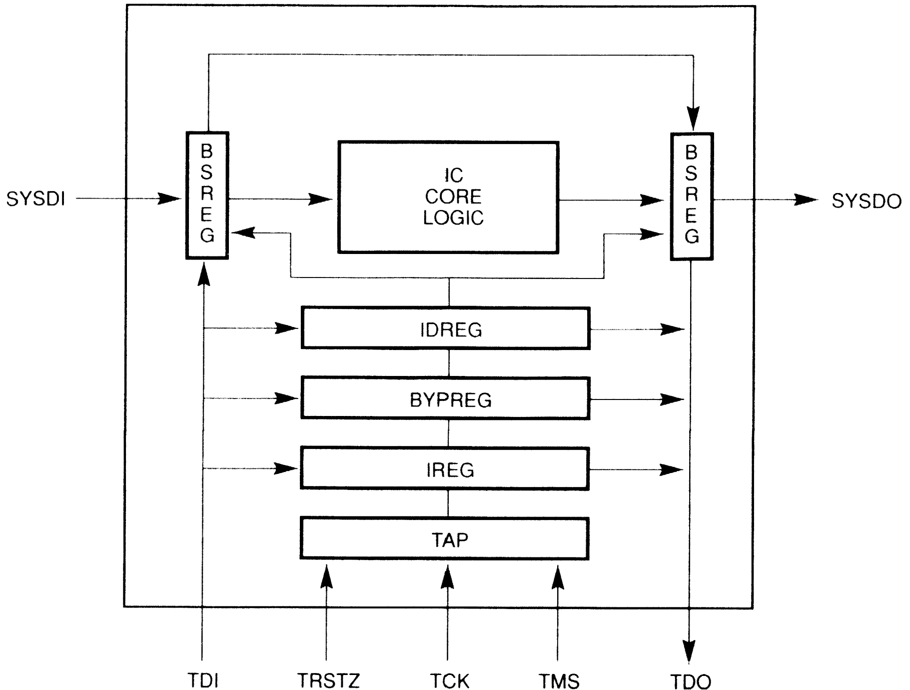
The test architecture is expandable to support additional IC-resident test features, such as internal scan and built-in self-test. The following description provides an overview of the IEEE 1149.1 boundary scan architecture and the test access port. This overview provides background information to support the other TI-FPGA boundary scan application notes. For more information on IEEE 1149.1, refer to IEEE standard 1149.1-1990, available from the IEEE by calling 800-678-IEEE.

8.5.1 Integrated Circuit Level View of 1149.1

An IC incorporating the 1149.1 boundary scan test architecture is shown in Figure 8-41. The test architecture consists of a test access port (TAP), instruction register (IREG), scan bypass register (BYPREG), identification register (IDREG), and boundary scan register (BSREG) consisting of input and output sections. The TAP, IREG, BYPREG, and BSREG are required components of the 1149.1 architecture; the IDREG is optional.

† Contributed by Lee Whetsel, Senior Member Technical Staff, Test Technology Center, Texas Instruments Incorporated.

Figure 8-41. IEEE 1149.1 Architecture



The TAP provides the control interface for serially accessing the shift registers within the test architecture. The instruction register is a shift register that provides storage for test commands input into the architecture.

The boundary scan register is a shift register that provides the input and output test circuitry required for testing the wiring interconnects between ICs on a printed circuit board.

The bypass register is a 1-bit shift register that provides an abbreviated scan path through the IC when boundary testing is not being performed. The optional identification register is a 32-bit shift register that provides information about the IC, such as manufacturer, device type, and version codes.

Control to access the shift registers within the architecture is input to the TAP via the test mode select (TMS) and test clock (TCK) pins. The TAP also has input for an optional test reset (TRSTZ) pin. Serial data is input to the selected shift register via the test data input (TDI) pin and serial data is output from the selected shift register via the test data output (TDO) pin.

When the test architecture is disabled, the IC operates in its normal mode, i.e. the system data input (SYSDI) to and system data output (SYSDO) from the IC core logic occurs without interruption. However, when the test architecture is enabled for boundary testing, the normal operation of the IC is interrupted while test data is input to and output from the IC boundary via the boundary scan register.

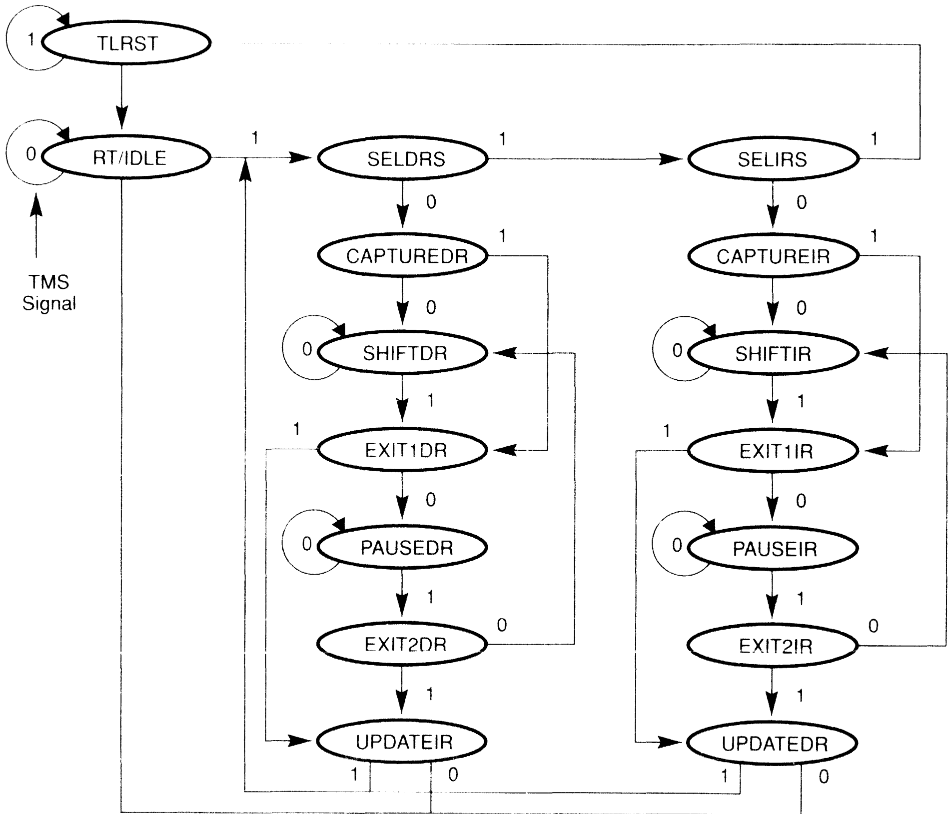
8.5.2 Test Access Port

The TAP is a 16-state finite state machine that receives external test clock (TCK) and test mode select (TMS) inputs and outputs internal control to enable serial data to be scanned into and out of the test architecture via the test data input (TDI) and test data output (TDO) pins. TCK provides the clock input for the TAP state machine and TMS provides the control input required to transition the TAP through its predefined test control states. In addition to the required TMS and TCK inputs, the TAP includes an optional input for a test reset (TRSTZ) input. The TRSTZ input provides a low active, asynchronous reset for the TAP and 1149.1 test architecture.

8.5.2.1 TAP State Diagram and Operation Modes

The state diagram shown in Figure 8-42 details the operation of the TAP in response to TMS signal input. In the state diagram all state transitions occur on the rising edge of TCK and the action of the state occurs on the falling edge of TCK. The action and next state transition of each TAP state is described in the following list.

Figure 8-42. 1149.1 TAP State Diagram



The run test idle state (RT/IDLE) provides an idle state for the TAP to reside in whenever instruction or data scan operations are not being performed. The RT/IDLE state is also used as a state where built-in-self-test operations are executed. Next states from the TLRST state are selectable to be the TLRST or RT/IDLE states.

The test logic reset state (TLRST) resets the test architecture and prevents test functions from accidentally becoming enabled during normal operation of the IC. Next states from the TLRST state are selectable to be the TLRST or RT/IDLE states.

The capture data register state (CAPTUREDR) allows the selected data register to parallel load test data. Next states from the CAPTUREDR state are selectable to be either the SHIFTDR or EXIT1DR states.

The shift data register state (SHIFTDR) allows the selected data register to shift data from the TDI input to the TDO input for a specified number of TCKs. The first serial data packet shifted out of the data register via TDO is the captured parallel test response data. The last serial data packet shifted into the data register via TDI is the new parallel test stimulus data to be output from the data register. Next states from the SHIFTDR state are selectable to be either the SHIFTDR or EXIT1DR states.

The exit1 data register scan state (EXIT1DR) provides a primary exit from the data scan mode. Next states from the EXIT1DR state are selectable to be either the PAUSEDR or UPDATEDR states.

The pause data register scan state (PAUSEDR) enables a data scan operation to be suspended for any number of TCKs. Next states from the PAUSEDR state are selectable to be either the PAUSEDR state or EXIT2DR states.

The exit2 data register scan state (EXIT2DR) provides a secondary exit from the data scan mode. Next states from the EXIT2DR state are selectable to be either the SHIFTDR or UPDATEDR states. The update data register state (UPDATEDR) allows the test stimulus data shifted into the data register to be parallel output from the data register. Next states from the UPDATEDR state are selectable to be either the run-test/idle (RT/IDLE) state or the select data register scan (SELDRS) state.

The capture instruction register state (CAPTUREIR) allows the instruction register to parallel load test status. Next states from the CAPTUREIR state are selectable to be either the SHIFTIR or EXIT1IR states.

The shift instruction register state (SHIFTIR) allows the instruction register to shift data from the TDI input to the TDO output for a specified number of TCKs. The first serial data packet shifted out of the instruction register via TDO is the captured parallel status information. The last serial data packet shifted into the instruction register via TDI is the new instruction code to be parallel output from the instruction register. Next states from the SHIFTIR state are selectable to be either the SHIFTIR or EXIT1IR states.

The exit1 instruction register scan state (EXIT1IR) provides a primary exit from the instruction scan mode. Next states from the EXIT1IR state are selectable to be either the PAUSEIR or UPDATEIR states.

The pause instruction register scan state (PAUSEIR) enables an instruction scan operation to be suspended for any number of TCKs. Next states from the PAUSEIR state are selectable to be either the PAUSEIR or EXIT2IR state.

The exit2 instruction register scan state (EXIT2IR) provides a secondary exit from the instruction scan mode. Next states from the EXIT2IR state are selectable to be either the SHIFTIR or UPDATEIR states.

The update instruction register state (UPDATEIR) allows the instruction code shifted into the instruction register to be parallel output from the instruction register. Next states from the UPDATEIR state are selectable to be either the run-test/idle (RT/IDLE) state or the select data register scan (SELDRS) state.

8.5.2.2 TAP operation modes

The TAP can operate in one of the five operation modes. The following descriptions detail the operation of the TAP and related shift registers during each of the five modes.

- Forced reset mode

  In response to TMS input, the TAP can enter into a test logic reset (TLRST) state. This test operation is used to force the test logic into a disabled condition so that it cannot interfere with the normal operation of the host IC.

  When testing is not required, the IC functions normally and the 1149.1 test logic is forced into a reset condition by transitioning the TAP into its TLRST state. Forcing a reset condition prevents the test logic from inadvertently becoming enabled to interfere with the normal operation of the IC. The 1149.1 standard describes two methods of forcing a reset condition.

  The first method of forcing a reset condition is to synchronously transition the TAP into its TLRST state by setting and leaving the TMS input high while the TCK input free-runs. This first method allows forcing the test logic into reset using only the four required test bus pins. The second method of forcing a reset condition is to asynchronously move the TAP into its TLRST state by setting and holding the optional TRSTZ input low.

- Idle mode

  In response to TMS input, the TAP can enter into a run test/idle (RT/IDLE) state with no test operation enabled. This test operation results in no test activity.

  Description: When the test logic is not required to be in a forced reset condition and when testing or scanning operations are currently not being performed, the TAP can be transitioned into and left in its RT/IDLE state for any length of time.

- Run test mode

  In response to TMS input, the TAP can enter into a run test/idle (RT/IDLE) state with a test operation enabled. This operation results in the execution of a user-specified test function defined by the current instruction in the instruction register.

  When a self-test instruction has been input to the instruction register, the TAP can be transitioned into the RT/IDLE state for a predetermined number of TCK cycles. When the TAP enters the RT/IDLE state the self-test starts and continues while the TAP remains in the RT/IDLE state. The self-test is stopped by transitioning out of the RT/IDLE state.

- Instruction scan mode

  In response to TMS input, the TAP can enter into an instruction scan operation. This test operation is used to shift serial instruction data into and out of the instruction register via TDI and TDO.

  When instruction data is to be input to the instruction register of the test architecture, the TAP receives control on the TMS pin to enable data to be shifted into and out of the instruction register via the TDI and TDO pins. The instruction scan mode is comprised of the subset modes or states as shown in the TAP diagram of Figure 8-44 and described below.

- Data scan mode

  In response to TMS input, the TAP can enter into a data scan operation. This test operation is used to shift serial test data into and out of a selected data register via TDI and TDO.

  When test data is to be input to a selected data register of the test architecture, the TAP receives control on the TMS pin to enable data to be shifted into and out of the selected data register via the TDI and TDO pins. The data scan mode is comprised of the subset modes or states as shown in the TAP diagram of Figure 8-44 and described below.

8.5.3 Instruction Register

The instruction register is a multiple stage shift register used to store a test instruction shifted into the architecture during a TAP instruction scan operation. During an instruction scan operation, serial data is shifted through the instruction register from TDI to TDO. The stored instruction selects one of the data scan registers to be connected between TDI and TDO for scan access, and may also enable a test operation.

8.5.3.1 Instruction Codes

The 1149.1 standard defines seven test instructions. Of the seven instructions, three are required and four are optional. In addition, two new optional instructions have been developed for future inclusion in the 1149.1 standard. The following is a description of each test instruction, starting with the three required instructions and following with the optional instructions.

- BYPASS Instruction

  The required BYPASS instruction allows the IC to remain in a functional mode and connects the bypass register between TDI and TDO. The BYPASS instruction allows serial data to be transferred through the IC from TDI to TDO without affecting the operation of the IC. The bit code of this instruction is defined by the 1149.1 standard as "all ones".

- SAMPLE/PRELOAD Instruction

  The required SAMPLE/PRELOAD instruction allows the IC to remain in its functional mode and selects the boundary scan register to be connected between TDI and TDO. During this instruction, the boundary scan register can be accessed via a data scan operation, to take snapshot samples of the data entering and leaving the IC. This instruction is also used to preload test data into the boundary scan register prior to loading an EXTEST instruction. The bit code for this instruction is defined by the user.

- EXTEST Instruction

  The required EXTEST instruction places the IC into an external boundary test mode and selects the boundary scan register to be connected between TDI and TDO. During this instruction, the boundary scan register is accessed to drive test data off-chip via the boundary outputs and receive test data off-chip via the boundary inputs. The bit code of this instruction is defined by the 1149.1 standard to be "all zeros".

- INTEST Instruction

  The optional INTEST instruction places the IC in an internal boundary test mode and selects the boundary scan register to be connected between TDI and TDO. During this instruction, the boundary scan register is accessed to drive test data on-chip via the boundary inputs and receive test data on-chip via the boundary outputs. The bit code of this instruction is defined by the user.

- RUNBIST Instruction

  The optional RUNBIST instruction places the IC into a self-test mode, enables a comprehensive self-test of the IC's core logic, and selects a user-specified data register to be connected between TDI and TDO.

During this instruction, the boundary outputs are controlled so that they cannot interfere with neighboring ICs during the RUNBIST operation. Also, the boundary inputs are controlled so that external signals cannot interfere with the RUNBIST operation. The bit code of this instruction is defined by the user.

- CLAMP Instruction

  The optional CLAMP instruction sets the outputs of an IC to logic levels determined by the contents of the boundary scan register and selects the bypass register to be connected between TDI and TDO. The contents of the boundary scan register can be preset prior to loading this instruction via a SAMPLE/PRELOAD instruction. During this instruction, data can be shifted through the bypass register from TDI to TDO without affecting the condition of the outputs. The bit code of this instruction is defined by the user.

- HIGHZ Instruction

  The optional HIGHZ instruction sets the outputs of an IC to a disabled state and selects the bypass register to be connected between TDI and TDO. The effect of this instruction is totally defined by the instruction and no prescanning of any data register is required. During this instruction, data can be shifted through the bypass register from TDI to TDO without affecting the condition of the IC outputs. The bit code of this instruction is defined by the user.

- IDCODE Instruction

  The optional IDCODE instruction allows the IC to remain in its functional mode and selects an identification register to be connected between TDI and TDO. The identification register shown in Figure 8-41 is a 32-bit shift register containing information regarding the IC manufacturer, device type, and version code.

  Accessing the identification register does not interfere with the operation of the IC. Also, access to the identification register should be immediately available, via a TAP data scan operation, after power up of the IC or after the TAP has been reset using the optional TRST pin. The bit code of this instruction is defined by the IC manufacturer.

- USERCODE Instruction

  The optional USERCODE instruction allows the IC to remain in its functional mode and selects a user register to be connected between TDI and TDO. The user register is a 32-bit shift register containing user defined information about the IC.

  The user register can be realized by reusing the IDREG to preload and shift out user defined information. Accessing the user register does not interfere with the operation of the IC. The bit code of this instruction is defined by the user.

8.5.4 Bypass Register

The bypass register is a single stage shift register that serves to abbreviate the scan path through the architecture during a TAP data scan operation. When selected by the current instruction, serial data is shifted through the bypass register from TDI to TDO during a TAP data scan operation.

8.5.5 Identification Register

The identification register is a 32-bit shift register that can be serially accessed to identify ICs manufacturer, part type, and revision code. When selected by the current instruction, serial data is shifted through the 32-bit identification register from TDI to TDO during a TAP data scan operation.

8.5.6 Boundary Scan Register

The boundary scan register consists of a series of boundary scan cells. Each boundary scan cell is associated with an IC input or output pin or a control signal regulating the state of an IC output pin, (i.e. 3-state control signal). When selected by the current instruction, serial data is shifted through the boundary scan register from TDI to TDO during a TAP data scan operation. The test function performed by the boundary scan register is determined by the instruction loaded in the instruction register.

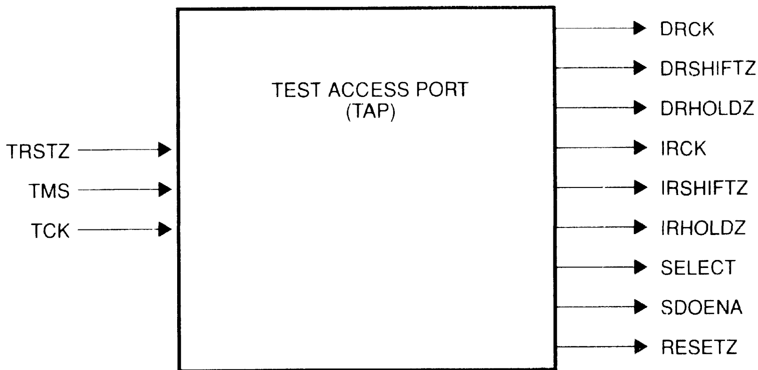
8.5.7 IEEE 1149.1 Boundary Scan Library Components

A library of test cell components, containing all the test cell components required to construct a boundary scan test structure, facilitates implementation of the 1149.1 boundary scan standard in FPGA devices.

8.5.7.1 TAP Macro

The TAP macro is the most important component of the 1149.1 test architecture. A good understanding of the TAP and its operation is essential to the design of 1149.1 compatible test structures. The library symbol of the TAP component is shown in Figure 8-43.

Figure 8-43. Test Access Port (TAP)



The TAP symbol has three inputs TRSTZ, TMS, and TCK; and nine outputs: DRCK, DRSHIFTZ, DRHOLDZ, IRCK, IRSHIFTZ, IRHOLDZ, SELECT, SDOENA, and RESETZ.

TAP input signals are externally received via input buffers coupled to package pins. The TMS and optional TRSTZ input signals require a pullup resistor so that the TAP receives logic high level inputs if the pins are not externally driven. TAP output signals are internally routed to control other components within the boundary scan architecture.

The TAP macro contains the following input and output signals.

- TRSTZ

  The Test Reset input is an optional active-low signal used to asynchronously reset the TAP controller.

- TMS

  The Test Mode Select input is a required signal that controls the operation of the TAP controller.

- TCK

  The Test Clock input is a required signal that provides the clock input for the TAP controller.

- DRCK

  The Data Register Clock output becomes active during a data register scan operation to parallel load then shift data through a selected data register from TDI to TDO.

- DRSHIFTZ

  The Data Register Shift output is high during the first DRCK of a data register scan operation to allow the selected data register to capture or preload test data.

- DRHOLDZ

  The Data Register Hold output is set low during data register scan operations to cause the output latches of the scan cells in the selected data register to hold their present state.

- IRCK

  The Instruction Register Clock output becomes active during an instruction register scan operation to parallel load then shift data through the instruction register from TDI to TDO.

- IRSHIFTZ

  The Instruction Register Shift output is high during the first IRCK of an instruction register scan operation to allow the instruction register to capture or preload with status data.

- IRHOLDZ

  The Instruction Register Hold output is set low during instruction register scan operations to cause the output latches of the scan cells in the instruction register to hold their present state.

- SELECT

  The SELECT output controls a multiplexer which inputs serial data to the TDO output pin of the IC during instruction and data scan operations.

- SDOENA

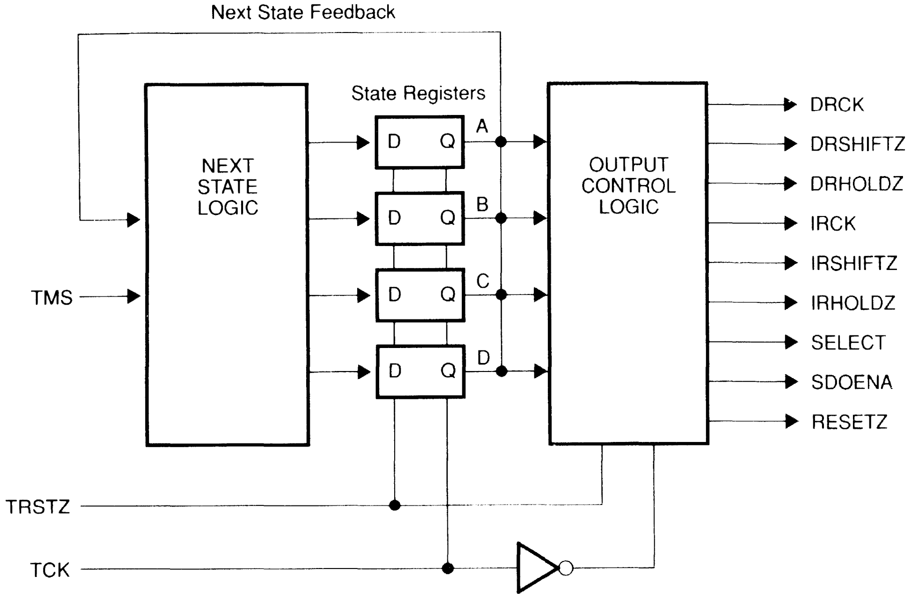
  The Serial Data Output enable output is an active-low signal that enables the TDO output buffer during instruction and data scan operations.

- RESETZ

  The RESETZ output is an active-low signal that resets the instruction register when the TAP enters its test logic reset state.

A block diagram of the TAP component is shown in Figure 8-44. The TAP is a state machine consisting of three sections; next state logic, state registers, and output control logic.

Figure 8-44. TAP Block Diagram



The next state logic provides the Boolean equations for the TAP to operate according to the state diagram of Figure 8-42. The next state logic receives input from the external TMS signal and the state register outputs (A,B,C, D).

The state register, which consists of four flip-flops and provides the state storage for the controller, operates on the rising edge of TCK.

The output control logic consists of state decode logic and flip-flops, which operate on the falling edge of TCK.

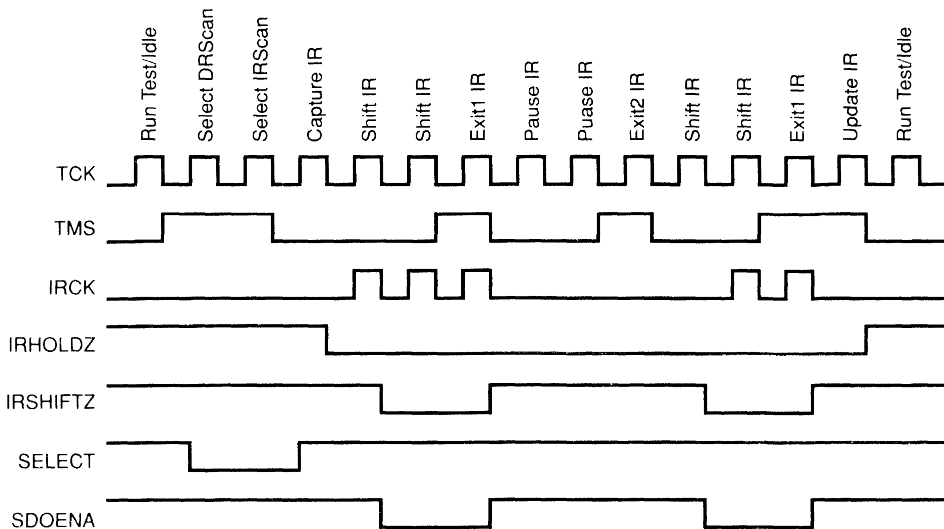
The optional TRSTZ input is an active-low asynchronous reset signal that forces the state register and output control flip-flops to an initialized state. Table 8-13 defines the logic level of each state-register flip-flop for each of the states in the TAP16-state diagram.

Table 8-13. TAP State Table

TAP State	Binary	Hex
	DCBA	
Exit2 DR	0000	0
Exit1 DR	0001	1
Shift DR	0010	2
Pause Dr	0011	3
Select IR	0100	4
Update DR	0101	5
Capture DR	0110	6
Select DR	0111	7
Exit2 IR	1000	8
Exit1 IR	1001	9
Shift IR	1010	A
Pause IR	1011	B
Run Test/Idle	1100	C
Update IR	1101	D
Capture IR	1110	E
Test Logic Reset	1111	F

Figure 8-45 illustrates the TMS input sequence that causes the TAP to execute an instruction scan operation of a 4-bit instruction register.In response to the TMS input, the TAP exists from the Run Test/Idle state and enters into the CAPTUREIR state via the SELECT-DR and SELECT-IR states. Upon entering the SHIFTIR state from the CAPTUREIR state, the instruction register preloads or captures status data, since IRSHIFTZ is high and the first IRCK is present. Also the output latches of the instruction register are latched by the IRHOLDZ signal being set low. On the second and third IRCKs, the instruction register shifts data since IRSHIFTZ is low.

Figure 8-45. TAP Instruction Register Scan Timing Diagram



NOTE: DRCK = low, DRSHIFTZ = high, and DRHOLDZ = high

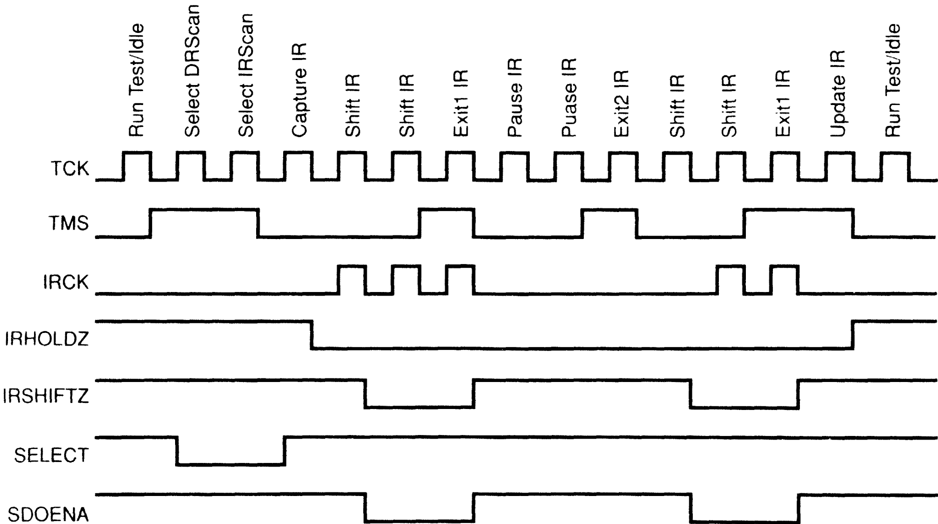
To illustrate the pausing capability during instruction-register scan operations, the TMS sequence causes the TAP to transition from the SHIFTIR state into the PAUSEIR state via the EXIT1IR state. In the PAUSEIR state the shifting of data is suspended since the IRCK is gated off. From the PAUSEIR state the TAP transitions back into the SHIFTIR state via the EXIT2IR state. The shifting of data resumes for two IRCKs while the TAP is in the SHIFTIR state. The instruction scan operation is terminated when the TAP transitions into the UPDATEIR state via the EXIT1IR state. In the UPDATEIR state the IRHOLDZ signal is set high to allow the data shifted into the instruction register to be output from the instruction register output latches. From the UPDATEIR state the TAP transitions into the Run Test/Idle state.

While the TAP is in the SHIFTIR state, the SDOENA signal is low to enable the TDO output buffer. During instruction scan operations, the SELECT signal is high to select the serial output of the instruction register to be output on TDO. When instruction-register scan operations are in progress; DRCK is low, DRSHIFTZ is high and DRHOLDZ is high.

Figure 8-46 illustrates the TMS input sequence that causes the TAP to execute a data-register scan operation. In response to the TMS input, the TAP exits from the Run Test/Idle state and enters into the CAPTUREDR state via the SELECT-DR state.

Upon entering the SHIFTDR state from the CAPTUREDR state, the selected data register preloads or captures test data, since DRSHIFTZ is high and the first DRCK is present. Also the output latches of the data register are latched by the DRHOLDZ signal being set low. On the second and third DRCKs, the data register shifts data since DRSHIFTZ is low.

Figure 8-46. TAP Data-Register Scan Timing Diagram



NOTE: DRCK = low, DRSHIFTZ = high, and DRHOLDZ = high

To illustrate the pausing capability during data-register scan operations, the TMS sequence causes the TAP to transition from the SHIFTDR state to the PAUSEDR state via the EXIT1DR state. In the PAUSEDR state the data shifting is suspended as the DRCK is gated off. From the PAUSEDR state the TAP transitions back to the SHIFTDR state via the EXIT2DR state. Data shifting resumes for two DRCKs while the TAP is in the SHIFTDR state.

The data-register scan operation is terminated when the TAP transitions into the UPDATEDR state via the EXIT1DR state. In the UPDATEDR state the DRHOLDZ signal is set high to allow the data shifted into the data register to

be output from the data-register output latches. From the UPDATEDR state the TAP transitions into the Run Test/Idle state.

While the TAP is in the SHIFTDR state, the SDOENA signal is low to enable the TDO output buffer. During data scan operations, the SELECT signal is low to select the serial output of the selected data register to be output on TDO. When data register scan operations are in progress; IRCK is low, IRSHIFTZ is high and IRHOLDZ is high.

Figure 8-47 and Table 8-14 illustrate the timing of the RESETZ output from the TAP. At the beginning of the diagram the TMS input is high and the TAP is in the test logic reset state with the RESETZ output set low. Setting the TMS input low causes the TAP to exit from the test logic reset state and enter into the Run Test/Idle state. The RESETZ output goes high on the falling edge of TCK after the Run Test/Idle state is entered. The TAP can be set back into the test logic reset state by setting the TMS input high. The RESETZ output is set low on the falling edge of the TCK after the test logic reset state has been entered. The state of the other TAP output signals during the test logic reset state is shown in Figure 8-47 and Table 8-14.

Figure 8-47. TAP Test Logic Reset Timing Diagram

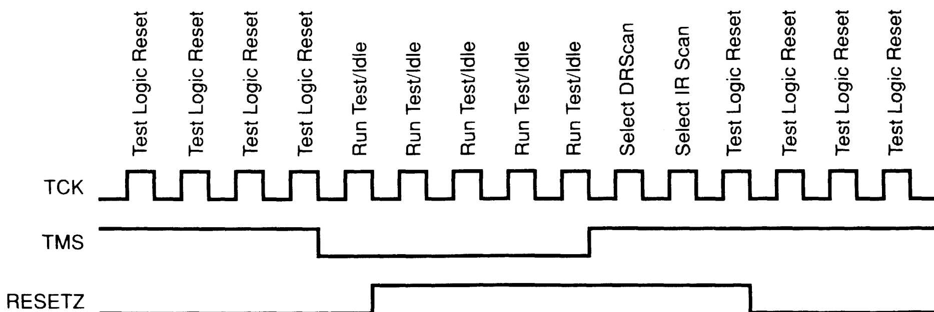
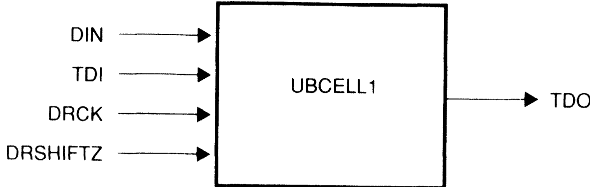


Table 8-14. TAP Outputs During Test Logic Reset State

Output	State
IRCLCK	Low
DRCLCK	Low
IRHOLDZ	Low
DRHOLDZ	Low
IRSHIFTZ	High
DRSHIFTZ	High
SELECT	High
SDOENA	High

The unidirectional boundary cell 1 (UBCELL1) shown in Figure 8-48 can only be used with unidirectional input boundary signals since it can only observe, not control, signal data. UBCELL1 supports the required EXTEST and SAMPLE/PRELOAD boundary test instructions, but does not support the optional INTEST instruction. UBCELL1 only uses two modules, thus this cell can be used to create input boundary register sections with a reduced module count. UBCELL1 has four inputs (DIN, TDI, DRCK, and DRSHIFTZ) and one output (TDO).

Figure 8-48. Unidirectional Boundary Cell 1



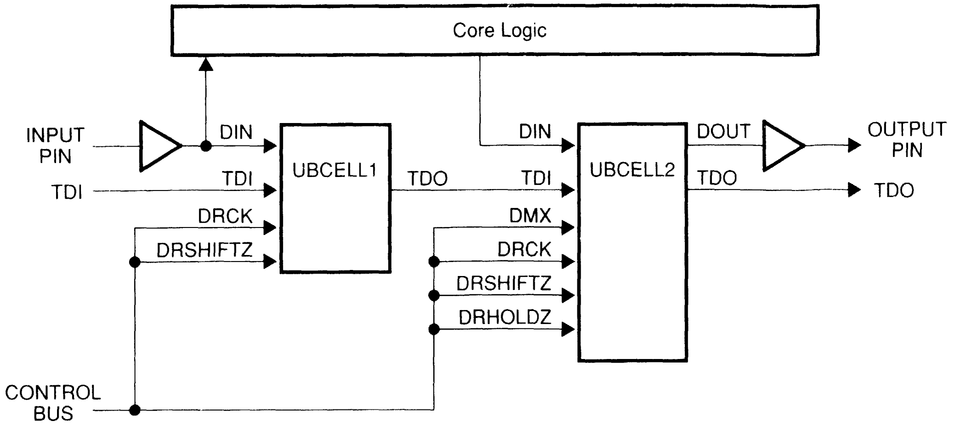
The truth table in Table 8-15 describes the function of UBCELL1. When a rising edge occurs on DRCK while DRSHIFTZ is high, the data present on DIN is loaded into the UBCELL1 flip-flop and output on TDO. When a rising edge occurs on DRCK while DRSHIFTZ is low, the data present on TDI is loaded into the UBCELL1 flip-flop and output on TDO.

Table 8-15. Unidirectional Boundary Cell 1 Truth Table

Function	Inputs		Output
	DRCK	DRSHIFTZ	TDO
Load Data	↑	1	DIN
Shift Data	↑	0	TDI

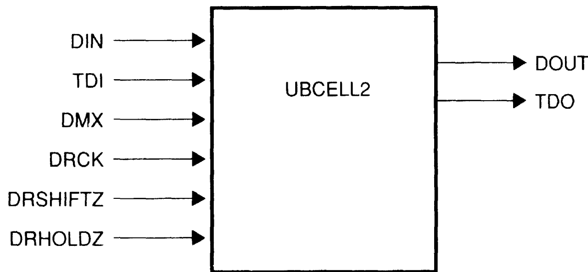
Figure 8-49 shows an example boundary scan register consisting of a UBCELL1 and UBCELL2. UBCELL1 and UBCELL2 are serially connected via their TDI and TDO data bus signals and parallel connected to the DRCK, DRSHIFTZ, DRHOLDZ and DMX control bus signals. Since UBCELL1 does not have output test circuitry, it does not require a connection to the DMX and DRHOLDZ control bus signals. UBCELL1 and UBCELL2 form a two-bit boundary scan register between the IC input and output pins and the core logic.

Figure 8-49. Boundary Register Using UBCELL1



The unidirectional boundary cell 2 (UBCELL2) shown in Figure 8-50 is designed to be used with unidirectional input or output boundary signals. The UBCELL2 cell is also used with signals that are used to control the state of output buffers; i.e., 3-state control signals. In test mode, UBCELL2 can observe and control the boundary signal. UBCELL2 supports the required EXTEST and SAMPLE/PRELOAD boundary test instructions and the optional INTEST instruction. UBCELL1 has six inputs (DIN, TDI, DMX, DRCK, DRSHIFTZ, and DRHOLDZ) and two outputs (TDO and DOUT).

Figure 8-50. Unidirectional Boundary Cell 2



The function of UBCELL2 is shown in Table 8-16.

Table 8-16. Unidirectional Boundary Cell 2 Truth Table

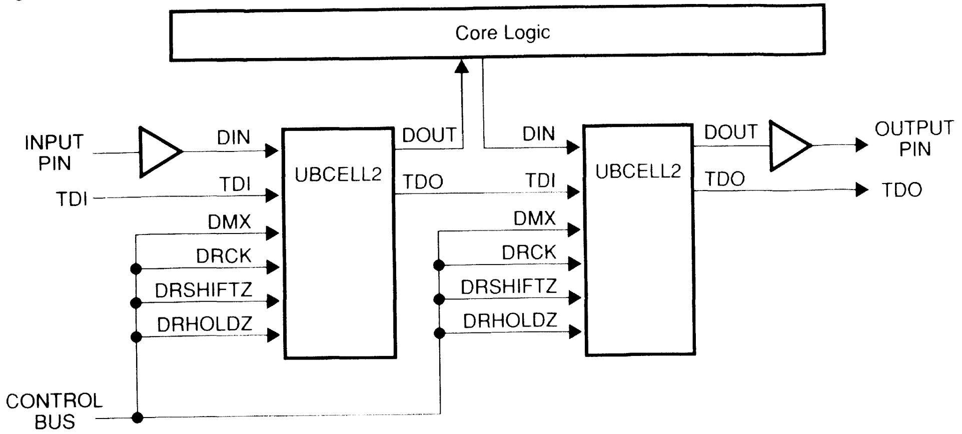
Function	Inputs				Outputs	
	DRCK	**DRSHIFTZ**	**DRHOLDZ**	**DMX**	**DOUT**	**TDO**
Normal Mode	0	1	1	0	DIN	FFQ
Load Data nm	↑	1	0	0	DIN	DIN
Shift Data nm	↑	0	0	0	DIN	FFQ
Update Data nm	↓	1	0 to 1	0	DIN	FFQ
Test Mode	0	1	1	1	FFQ	FFQ
Load Data tm	↑	1	0	1	LQ	DIN
Shift Data tm	↑	0	0	1	LQ	FFQ
Update Data tm	↓	1	0 to1	1	FFQ	FFQ

NOTE: FFQ = shift register output; LQ = latch output; nm = normal mode; tm = test mode.

While DMX is low the data on DIN transfers to DOUT. When a rising edge occurs on DRCK and while DRSHIFTZ is high and DRHOLDZ and DMX are low, the data present on DIN is loaded into the UBCELL2's flip-flop and output on TDO. When a rising edge occurs on DRCK and while DRSHIFTZ, DRHOLDZ and DMX are low, the data present on TDI is loaded into the UBCELL2's flip-flop and output on TDO. While DMX is high and DRHOLDZ is low, the data output on DOUT is latched and held during DRCK activations. While DMX is high and when DRHOLDZ transitions from a low to a high, the data output on DOUT becomes the new value stored in the UBCELL2's flip-flop.

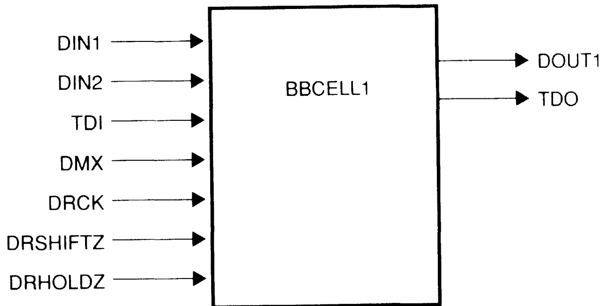
An example boundary scan register consisting of two UBCELL2s is shown in Figure 8-51. The input UBCELL2 and output UBCELL2 are serially connected via their TDI and TDO data bus signals and parallel connected to the DRCK, DRSHIFTZ, DRHOLDZ and DMX control bus signals. The input UBCELL2 and output UBCELL2 form a two-bit boundary scan register between the IC's input and output pins and core logic.

Figure 8-51. Boundary Register Using UBCELL2



The bidirectional boundary cell 1 (BBCELL1) shown in Figure 8-52 is designed to be used with bidirectional type boundary signals. The BBCELL1 consists of a UBCELL1 and a UBCELL2. The UBCELL1 part of the BBCELL1 is used to observe the input signal of a bidirectional signal pair, and the UBCELL2 part is used to observe and control the output signal of a bidirectional signal pair.

Figure 8-52. Bidirectional Boundary Cell 1



BBCELL1 supports the required EXTEST and SAMPLE/PRELOAD boundary test instructions, but does not support the optional INTEST instruction since it cannot control the input signal of a bidirectional signal pair. BBCELL1 has seven inputs (DIN1, DIN2, TDI, DMX, DRCK, DRSHIFTZ, and DRHOLDZ) and two outputs (TDO and DOUT).

The function of BBCELL1 is shown by the truth table in Table 8-17. The UBCELL2 part of the BBCELL1 is associated with the DIN1 input and DOUT1 output. The UBCELL1 part is associated DIN2 input. Both UBCELL1 and UBCELL2 receive input from the control inputs. The UBCELL1 and UBCELL2 within the BBCELL1 respond to the control inputs as described in their truth tables. During shift operations, UBCELL2 and UBCELL1 form a two-bit shift register.

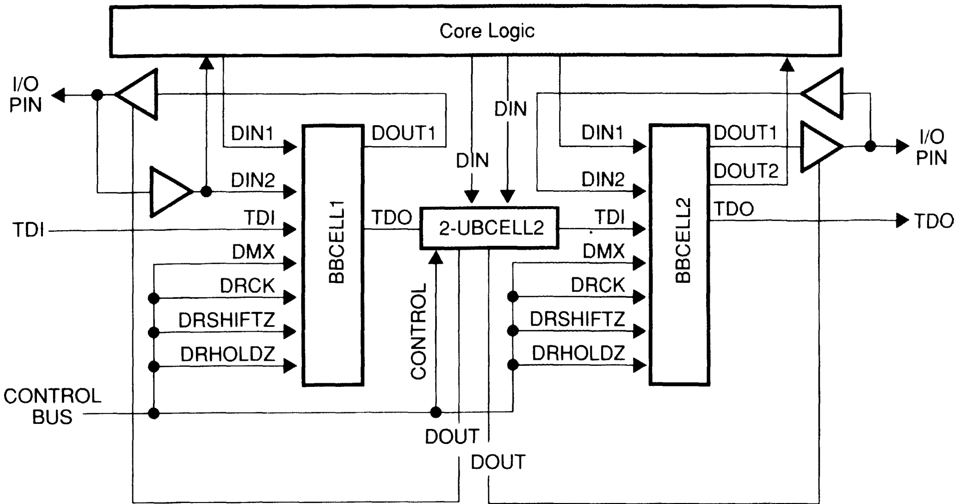
Table 8-17. Bidirectional Boundary Cell 1 Truth Table

Function	Inputs				Outputs	
	DRCK	DRSHIFTZ	DRHOLDZ	DMX	DOUT1	TDO
Normal Mode	0	1	1	0	DIN1	FFQ2
Load Data nm	↑	1	0	0	DIN1	DIN2
Shift Data nm	↑	0	0	0	DIN1	FFQ1
Update Data nm	↓	1	0 to 1	0	DIN1	FFQ2
Test Mode	0	1	1	1	FFQ1	FFQ2
Load Data tm	↑	1	0	1	LQ1	DIN2
Shift Data tm	↑	0	0	1	LQ1	FFQ1
Update Data tm	↓	1	0 to1	1	FFQ1	FFQ2

NOTE: FFQ1 = flip-flop 1 output; FFQ2 = flip-flop 2 output; LQ1 = Latch 1 output; LQ2 = Latch 2 output; nm = normal mode; tm = test mode.

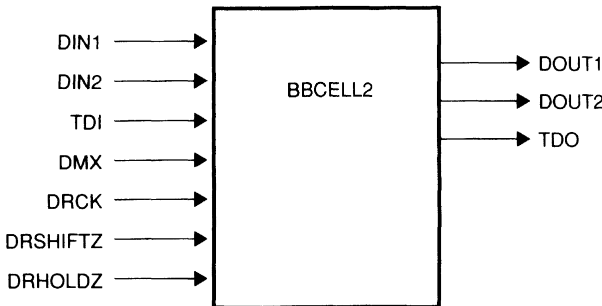
An example boundary scan register consisting of a BBCELL1, two UBCELL2s, and a BBCELL2 is shown in Figure 8-53. BBCELL1, the two UBCELL2, and BBCELL2 are serially connected via their TDI and TDO data bus signals and parallel connected to the DMX, DRCK, DRSHIFTZ, and DRHOLDZ control bus signals. The boundary scan cells form a six-bit boundary scan register between the IC's two bidirectional pins and the core logic, (BBCELL1 has two bits, each UBCELL2 has a bit, and BBCELL2 has two bits). During normal mode, the output control from the core logic transfers through the UBCELL2 cells, via their DIN inputs and DOUT outputs, to control the state of the output buffers. During test mode, the output control from the UBCELL2's DOUT output is determined by the value shifted in the cell.

Figure 8-53. Boundary Scan Register Using BBCELL1



The bidirectional boundary cell 2 (BBCELL2) shown in Figure 8-54 is designed to be used with bidirectional type boundary signals. The BBCELL2 consists of two UBCELL2 cells. One UBCELL2 is used to observe and control the input signal of a bidirectional signal pair, and the other UBCELL2 is used to observe and control the output signal of a bidirectional signal pair. BBCELL2 supports the required EXTEST and SAMPLE/PRELOAD boundary test instructions and the optional INTEST instruction. BBCELL2 has seven inputs (DIN1, DIN2, TDI, DMX, DRCK, DRSHIFTZ, and DRHOLDZ) and two outputs (TDO and DOUT).

Figure 8-54. Bidirectional Boundary Cell 2



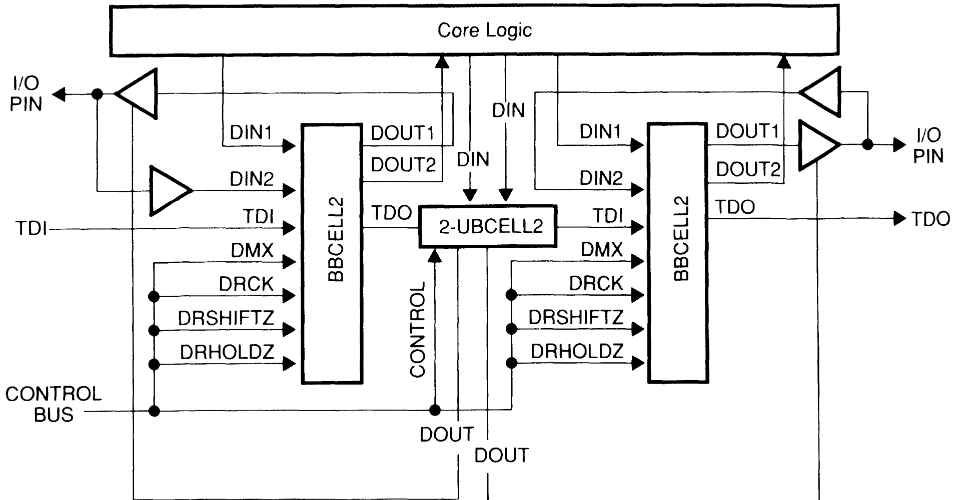
The function of BBCELL2 is described by the truth table in Table 8-18. One of the UBCELL2 cells in the BBCELL2 is associated with the DIN1 input and DOUT1 output, and the other UBCELL2 is associated with the DIN2 input and DOUT2 output. Both UBCELL2 cells receive input from the control inputs. Both UBCELL2 cells within the BBCELL2 respond to the control inputs as described in the UBCELL2 truth table.

Table 8-18. Bidirectional Boundary Cell 2 Truth Table

Function	**Inputs**				**Outputs**		
	DRCK	**DRSHIFTZ**	**DRHOLDZ**	**DMX**	**DOUT1**	**DOUT2**	**TDO**
Normal Mode	0	1	1	0	DIN1	DIN2	FFQ2
Load Data nm	↑	1	0	0	DIN1	DIN2	DIN2
Shift Data nm	↑	0	0	0	DIN1	DIN2	FFQ1
Update Data nm	↓	1	0 to 1	0	DIN1	DIN2	FFQ2
Test Mode	0	1	1	1	FFQ1	FFQ2	FFQ2
Load Data tm	↑	1	0	1	LQ1	LQ2	DIN2
Shift Data tm	↑	0	0	1	LQ1	LQ2	FFQ1
Update Data tm	↓	1	0 to1	1	FFQ1	FFQ2	FFQ2

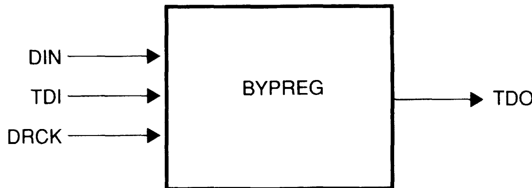
An example boundary scan register consisting of two BBCELL2 and two UBCELL2 cells is shown in Figure 8-55. The BBCELL2 and UBCELL2 cells are serially connected via their TDI and TDO data bus signals and parallel connected to the DMX, DRCK, DRSHIFTZ, and DRHOLDZ control bus signals. The boundary scan cells form a six-bit boundary scan register between the IC's two bidirectional pins and the core logic, (each BBCELL2 has two bits and each UBCELL2 has a bit). During normal mode, the output control from the core logic transfers through the UBCELL2 cells, via their DIN inputs and DOUT outputs, to control the state of the output buffers. During test mode, the output control from the UBCELL2's DOUT output is determined by the value shifted in the cell.

Figure 8-55. Boundary Register Using BBCELL2



The bypass register (BYPREG) shown in Figure 8-56 is designed to provide a single-bit shift register path between TDI and TDO when the BYPASS instruction is loaded and the TAP executes a data register scan operation. The BYPREG has three inputs (TDI, DRCK, and DRSHIFTZ) and one output (TDO).

Figure 8-56. Bypass Register (BYPREG)



The function of BYPREG is described in Table 8-19. When a rising edge occurs on DRCK while DRSHIFTZ is high, a logic zero is loaded into the BYPREG flip-flop and output on TDO. When a rising edge occurs on DRCK while DRSHIFTZ is low, the data present on TDI is loaded into the BYPREG flip-flop and output on TDO.

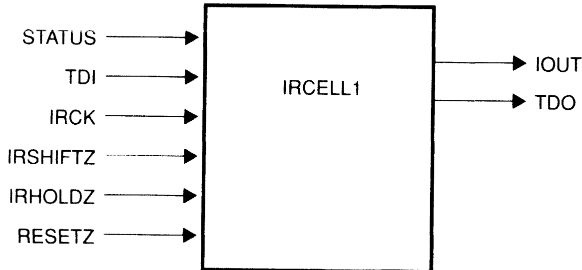
Table 8-19. Bypass Register (BYPREG) Truth Table

Function	Inputs		Output
	DRCK	DRSHIFTZ	TDO
Load Data	↑	1	0
Shift Data	↑	0	TDI

Two instruction-register cells are designed to simplify the construction of 1149.1 instruction registers, IRCELL1 and IRCELL2. The instruction cells are identical in operation with the exception that IRCELL1 is preset when the TAP outputs a RESETZ signal, and IRCELL2 is cleared when the TAP outputs a RESETZ signal. In most instances, the IRCELL1 will be used in instruction-register designs. However, if the user implements the optional identification register the IRCELL2 will be used at one or more instruction bit locations to provide a power up IDCODE instruction which is not all ones. By powering up with the IDCODE instruction loaded into the instruction register, immediate access to the IC identification register information is available via a data-register scan operation.

The instruction register cell 1 (IRCELL1) shown in Figure 8-57 is designed as a bit-slice element to be used in constructing 1149.1 instruction registers. IRCELL2 has seven inputs (Status, TDI, IRCK, IRSHIFTZ, IRHOLDZ, and RESETZ) and two outputs (IOUT and TDO).

Figure 8-57. Instruction Register Cell 1 (IRCELL1)



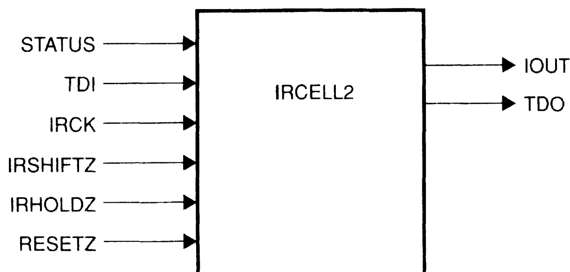
The function of the IRCELL1 is shown in Figure 8-57. In response to a low-level input on RESETZ, the IRCELL1's flip-flop is asynchronously initialized to a high logic state, independent of the other inputs. When a rising edge occurs on IRCK while IRSHIFTZ and RESETZ are high, the data present on the Status input is loaded into the IRCELL1's flip-flop and output on TDO. When a rising edge occurs on IRCK while IRSHIFTZ is low and RESETZ is high, the data present on TDI is loaded into the IRCELL1's flip-flop and output on TDO. While IRHOLDZ is low, the data output on IOUT is latched and held during IRCK activations. When IRHOLDZ transitions from a low to a high, the data output on IOUT becomes the new value stored in the IRCELL1 flip-flop.

Table 8-20. Instruction-Register Cell 1 (IRCELL1) Truth Table

Function	Inputs				Outputs	
	IRCK	**IRSHIFTZ**	**IRHOLDZ**	**RESETZ**	**IOUT**	**TDO**
Load Data	↑	1	0	1	LQ	Status
Shift Data	↑	0	0	1	LQ	TDI
Update Data	0	1	1	1	FFQ	FFQ
Reset	X	X	X	0	1	1

The instruction register cell 2 (IRCELL2) shown in Figure 8-58 is designed as a bit-slice element to be used in constructing 1149.1 instruction registers. IRCELL2 has seven inputs (Status, TDI, IRCK, IRSHIFTZ, IRHOLDZ, and RESETZ) and two outputs (IOUT and TDO).

Figure 8-58. Instruction Register Cell 2 (IRCELL2))



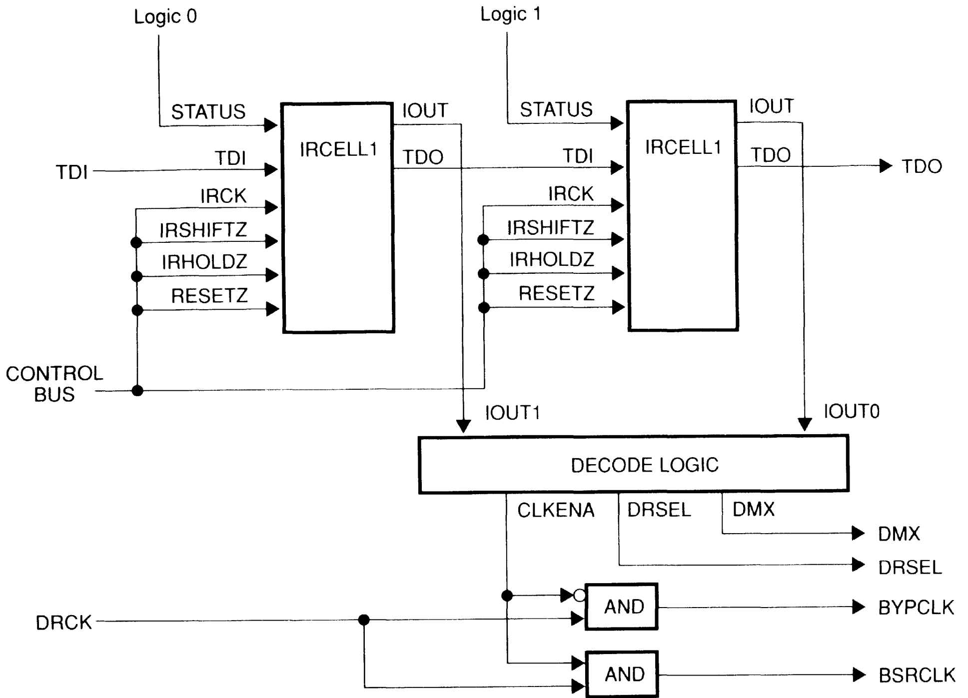
The function of the IRCELL2 is described in Table 8-21. In response to a low-level input on RESETZ, the IRCELL2 flip-flop is asynchronously initialized to a low logic state, independent of the other inputs. When a rising edge occurs on IRCK while IRSHIFTZ and RESETZ are high, the data present on the Status input is loaded into the IRCELL2 flip-flop and output on TDO. When a rising edge occurs on IRCK while IRSHIFTZ is low and RESETZ is high, the data present on TDI is loaded into the IRCELL2 flip-flop and output on TDO. While IRHOLDZ is low, the data output on IOUT is latched and held during IRCK activations. When IRHOLDZ transitions from a low to a high, the data output on IOUT becomes the new value stored in the IRCELL2 flip-flop.

Table 8-21. Instruction-Register Cell 2 (IRCELL2) Truth Table

Function	Inputs				Outputs	
	IRCK	IRSHIFTZ	IRHOLDZ	RESETZ	IOUT	TDO
Load Data	↑	1	0	1	LQ	Status
Shift Data	↑	0	0	1	LQ	TDI
Update Data	0	1	1	1	FFQ	FFQ
Reset	X	X	X	0	1	1

Figure 8-59 shows a instruction register consisting of two IRCELL1 cells.

Figure 8-59. Example Instruction Register



The first and second IRCELL1 cells are serially connected via their TDI and TDO data bus signals and parallel connected to the IRCK, IRSHIFTZ, IRHOLDZ and RESETZ control bus signals. The 1149.1 standard requires that the status input of the second IRCELL1 (least-significant cell) is connected to a logic high level. By choice, the status input of the first IRCELL1 (most-significant cell) is connected to a low logic level. The two IRCELL1 cells form a two-bit instruction register between the TDI and TDO pins.

In order to complete the design of the instruction register, the IOUT outputs from each IRCELL1 must be decoded to implement the instruction shifted in. For example purposes, an instruction decode logic section is included in Figure 8-59. The IOUT outputs from the instruction register are input to a decode logic block. The decode logic block decodes the two-bit instruction input into one of four 1149.1 instruction types, EXTEST, SAMPLE/PRELOAD, INTEST, and BYPASS. The EXTEST instruction is always decoded as all zeros, and the BYPASS instruction is always decoded as all ones.

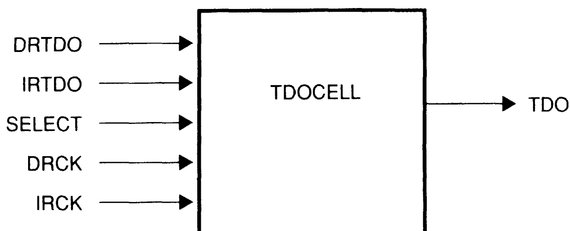
In response to the two-bit instruction input, the decode logic outputs control for a serial data multiplexer control (DMX) signal, a data register select (DRSEL) signal, and a data register clock enable (CLKENA) signal. The DMX signal is routed to the boundary register to enable boundary testing using EXTEST or INTEST instructions. The DRSEL signal is routed to a serial data multiplexer that selects the TDO output of the selected data register to be output on TDO, via the TDOCELL. The CLKENA signal enables either the boundary scan register clock (BSRCLK) or the bypass register clock (BYPCLK). The truth table shown in Table 8-22 illustrates which output control signal is enabled for each of the four types of 1149.1 instructions.

Table 8-22. Instruction Register Decode Truth Table

Test Instruction	Inputs		Outputs				
	IOUT1	**IOUT0**	**DMX**	**DRSEL**	**CLKENA**	**BYPCLK**	**BSRCLK**
Extest	0	0	1	1	1	0	DRCK
Sample/ Preload	0	1	0	1	1	0	DRCK
Intest	1	0	1	1	1	0	DRCK
Bypass	1	1	0	0	0	DRCK	0

The test data output cell (TDOCELL) shown in Figure 8-60 is designed to simplify the task of routing serial data off the IC via the TDO output pin. TDOCELL has five inputs (DRTDO, IRTDO, SELECT, DRCK, and IRCK) and one outputs (TDO). The DRTDO signal comes from the TDO output of the selected data register. The IRTDO signal comes from the TDO output of the instruction register. The SELECT signal comes from the TAP and is used to select either the DRTDO or IRTDO signals to be input to the TDOCELL flip-flop. The DRCK and IRCK signals come from the TAP and are used to clock the selected TDO input into the TDOCELL flip-flop on the falling edge of the clock. The TDO output of the TDOCELL outputs data to the IC TDO pin.

Figure 8-60. Test Data Output Cell (TDOCELL)



In the TDOCELL, the DRCK and IRCK signals are input to an OR gate and the output of the OR gate is input to the clock input of the flip-flip. When the TAP enables one of the clock signals, during a data or instruction scan operation, the other clock signal is set low enabling the selected clock signal to pass through the OR gate to clock the TDOCELL flip-flop. The truth table shown in Table 8-23 illustrates the function of the TDOCELL.

Table 8-23. Test Data Output Cell (TDOCELL) Truth Table

Function	Inputs			Output
	DRCK	**IRCK**	**Select**	**TDO**
Shift DRTDO	↓	0	0	DRTDO
Shift IRTDO	0	↓	1	IRTDO

The identification register 1 (IDREG1) shown in Figure 8-61 is a 32-bit shift register. IDREG1 has three inputs (TDI, DRCK, and DRSHIFTZ) and one output (TDO). The function of the IDREG1 is described in Table 8-24. When a rising edge occurs on DRCK while DRSHIFTZ is high, the identification data is loaded into the 32 IDREG1 flip-flops. When a rising edge occurs on DRCK while DRSHIFTZ is low, data is shifted from TDI, through the 32-bit shift register to TDO.

Figure 8-61. Identification Register 1 Macro (IDREG1)

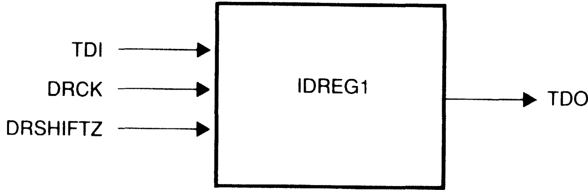


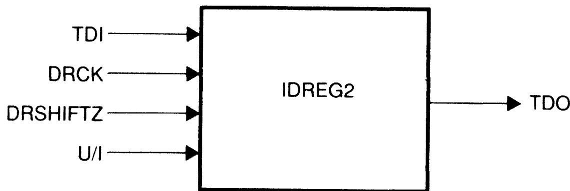
Table 8-24. Identification Register 1 Macro (IDREG1) Truth Table

Function	Inputs		Output
	DRCK	DRSHIFTZ	TDO
Load ID Data	↑	1	ID0 = 1
Shift ID Data	↑	0	ID1

IDREG1 meets the rules in the 1149.1 standard relating to the optional identification register. IDREG1 can be included in the 1149.1 architecture to provide information about the IC. The 32-bit shift register of the IDREG1 is broken down into fields. The least-significant bit (the one nearest TDO) must be wired to preload to a logic high level. The next least-significant 11 bits must be wired to preload with the manufacturers identification code. The next least-significant 16 bits must be wired to preload with the IC part number. The last four bits must be wired to preload with the IC revision number.

The identification register 2 (IDREG2) shown in Figure 8-62 is a 32-bit shift register. IDREG2 has four inputs (TDI, DRCK, DRSHIFTZ, and U/I) and one output (TDO). The function of the IDREG2 is described in Table 8-25.

Figure 8-62. Identification Register 2 Macro (IDREG2)



The only difference between IDREG1 and IDREG2 is that IDREG2 has an added User code/Identification code (U/I) input that allows the 32-bit shift register to output either the 32-bit identification code, as described in the IDREG1 section, or a 32-bit user code defined by the customer.

Table 8-25. Identification Register 2 Macro (IDREG2) Truth Table

Function	Inputs			Output
	DRCK	**DRSHIFTZ**	**U/I**	**TDO**
Load ID Data	↑	1	0	ID0 = 1
Shift ID Data	↑	0	0	ID1
Load UD Data	↑	1	1	UD0 = 1
Shift UD Data	↑	0	1	UD1

Figure 8-62 shows that when the U/I input is low the IDREG2 operates to output the 32-bit identification code. When the U/I input is high the IDREG2 operates to output the 32-bit user code. The U/I signal comes from the instruction register and is set high when a USERCODE instruction is loaded.

When a rising edge occurs on DRCK while DRSHIFTZ is high and U/I is low, the identification data is loaded into the 32 IDREG2 flip-flops. When a rising edge occurs on DRCK while DRSHIFTZ and U/I are high, the user data is loaded into the 32 IDREG2 flip-flops. When a rising edge occurs on DRCK while DRSHIFTZ is low, data is shifted from TDI, through the 32-bit shift register to TDO.

8.5.8 IEEE 1149.1 Test Architecture Design Options

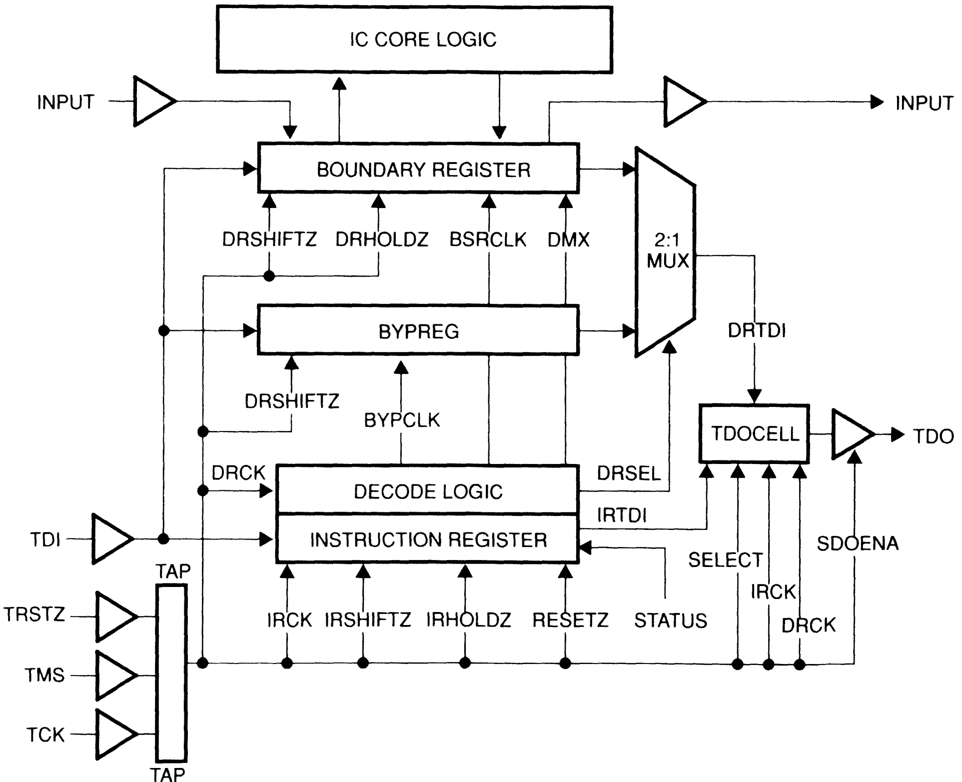
Two test interface macros (TIMs) are designed to enhance customer implementation of the 1149.1 boundary scan architecture in TI FPGAs. Within a single library macro TIMs provide all the test logic required to implement the boundary scan architecture, excluding the boundary scan register and, optionally, the identification register. TIMs are attractive for customers who are not experienced in 1149.1 design practices, since they provide a simple turn-key approach to installing boundary scan in their designs. Users who are familiar with 1149.1 may chose to implement the boundary scan architecture using the primitive test cell components provided in the FPGA library.

The following sections give examples of three 1149.1 design styles the customer can use to implement boundary scan in FPGA designs. The first design style illustrates how the experienced 1149.1 designer can create the boundary scan architecture using the basic 1149.1 building block components offered in the FPGA library. The next two design styles illustrate the use of the TIM1 and TIM2 circuits to simplify boundary scan insertion into FPGA designs.

8.5.8.1 Designing 1149.1 Using Library Components

An example implementation of an 1149.1 boundary scan architecture is shown in Figure 8-63. The test architecture consists of predefined components from the 1149.1 library including the TAP, TDOCELL, and BYPREG. The test architecture also consists of a 2:1 multiplexer, a user defined instruction register section, consisting of IRCELLS and decode logic, and a user defined boundary register section comprised of user selected boundary scan cells (UBCELL1, UBCELL2, BBCELL1, and BBCELL2). In this 1149.1 design example, the identification register is not implemented. The test-cell components illustrated in Figure 8-63 are described in detail in Section 8.5.

Figure 8-63. Example 1149.1 Test Architecture



The TAP receives input from the TRSTZ, TMS, and TCK test pins, and outputs control signals (IRCK, IRSHIFTZ, IRHOLDZ, RESETZ, DRCK, DRSHIFTZ, DRHOLDZ, SELECT, and SDOENA) to the boundary scan register, instruction register, BYPREG, TDOCELL, and TDO 3-state buffer.

The instruction register and decode logic receives the TDI signal, the IRCK, DRCK, IRSHIFTZ, IRHOLDZ and RESETZ control signals from the TAP, and status inputs. The instruction register and decode logic outputs a TDO signal and control signals (DRSEL, BYPCLK, BSRCLK, and DMX) to the 2:1 multiplexer, BYPREG, boundary scan register, and TDOCELL. The instruction register section consists of user-connected boundary instruction cells (IRCELL1, IRCELL2). The decode logic section consists of combinational logic gates that decode the instruction pattern shifted into the instruction register section. The minimum instruction patterns that must be decoded and included in the 1149.1 test architecture are the EXTEST, SAMPLE/PRELOAD, and BYPASS instructions.

The boundary scan register receives the TDI signal, the DRSHIFTZ and DRHOLDZ control signals from the TAP, the BSRCLK and DMX control signals from the instruction register, and data input from the IC core logic and input pins. The boundary scan register outputs a TDO signal to the 2:1 multiplexer and data signals to the IC core logic and output pins. The boundary scan register consists of user-connected boundary cells (UBCELL1, UBCELL2, BBCELL1, BBCELL2). A boundary cell must exist for each system input and output pin on the IC. Also a boundary cell must exist for each control signal that regulates 3-state output buffers.

The BYPREG receives the TDI signal, the DRSHIFTZ control signal from the TAP, and the BYPCLK signal from the instruction register. The BYPREG outputs a TDO signal to the 2:1 multiplexer.

The 2:1 multiplexer receives the TDO signals from the boundary scan register and BYPREG and the DRSEL control signal from the instruction register. The 2:1 multiplexer outputs a TDO signal to the TDOCELL.

The TDOCELL receives the TDO signal from the 2:1 multiplexer, the TDO signal from the instruction register, and control signals (SELECT, IRCK, and DRCK) from the TAP. The TDOCELL outputs a TDO signal to the IC TDO pin via the 3-state output buffer.

During instruction-register scan operations, the TAP outputs control to shift data through the instruction register and TDOCELL from the TDI input to the TDO output. The outputs of the decode logic are prevented from changing until the instruction scan operation has completed.

During boundary scan register scan operations, the TAP outputs control to shift data through the boundary scan register, 2:1 multiplexer, and TDOCELL from the TDI input to the TDO output. The outputs from boundary scan register cells associated with output buffers are prevented from changing state until the boundary data scan operation has completed.

During bypass register scan operations, the TAP outputs control to shift data through the BYPREG, 2:1 multiplexer, and TDOCELL from the TDI input to the TDO output.

During any scan operation, the data at TDI is input during the rising edge of TCK and the data at TDO is output on the falling edge of TCK. If the TAP TRSTZ input is low, the instruction register will be initialized with a BYPASS instruction, the TDI, TMS, and TCK signals will be ignored, and the TDO output buffer will be 3-state.

Note:

Performing a data-register scan operation after the TAP has been initialized, by either activating the TRSTZ pin or by scanning the TAP into its test logic reset, will cause data to be shifted through the BYPREG from the TDI input to TDO output pin.

8.5.8.2 Module Count Calculation

When designing the 1149.1 boundary scan architecture of Figure 8-63 into FPGAs, it is useful to know the number of modules the test logic requires. The following equation is provided to allow the designer to calculate the number of modules used in designing the 1149.1 architecture into an FPGA IC using the primitive 1149.1 test cells.

Module Count = TAP + IREG + BYPREG + 2:1 Mux + TDOCELL + Boundary Register

Where:

TAP = 65 modules
IREG = # Modules in IRCELLs + Decode Logic
IRCELL = 3 Modules
BYPREG = 2 modules
2:1 Mux = 1 Module
TDOCELL = 3 Modules
Boundary Register = # Modules in B-Cells

Module counts for each 1149.1 test cell can be found in Table 8-26 for a quick reference to the number of modules required to implement any 1149.1 test cell component and the TIM1 and TIM2 test interface circuits. It is suggested that an estimate of the number of modules required to implement 1149.1 in an FPGA be made early in the design flow, to allow the designer to clearly understand the impact of inserting the boundary scan test methodology

Table 8-26. Module Count Look-Up Table

Each UBCELL1 component requires 2 modules.
Each UBCELL2 component requires 4 modules.
Each BBCELL1 component requires 6 modules.
Each BBCELL2 component requires 8 modules.
Each IRCELL component requires 3 modules.
The BYPREG component requires 2 modules.
The TDOCELL component requires 3 modules.
The IDREG1 component requires 68 modules.
The IDREG2 component requires 102 modules.
The TAP component requires 65 modules.
The TIM1 circuit requires 79 modules.
The TIM2 circuit requires 90 modules.

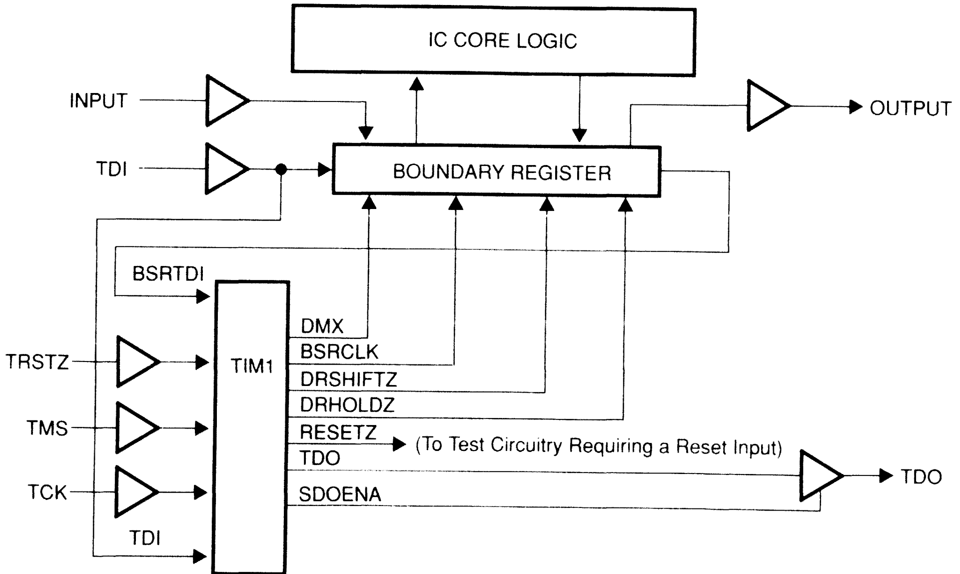
Assuming a design includes a minimum 2-bit instruction register (two IRCELLs), seven modules of decode logic, and a boundary scan register consisting of 40 UBCELL2 cells, the module count can be calculated as shown below.

Module Count = TAP + IREG + BYPREG + 2:1 Mux + TDOCELL + Boundary Register
= 65 + ((2 • 3) +7) + 2 + 1 + 3 + (40 • 4)
= 244 Modules

8.5.8.3 Designing 1149.1 Using a TIM1 Circuit

The test interface macro 1 (TIM1) shown in Figure 8-64 is designed to simplify the construction of 1149.1 test structures in FPGA devices. TIM1 is a predefined 1149.1 interface consisting of a TAP, a two-bit instruction register, a bypass register, and a TDOCELL. TIM1 includes the three required test instructions for an 1149.1 compliant boundary scan architecture, EXTEST, SAMPLE/PRELOAD, and BYPASS), along with the optional INTEST instruction. When the TIM1 is used to construct an 1149.1 test structure, the user only needs to design the boundary scan register and then connect the boundary scan register to the TIM1 circuit as shown in Figure 8-64. TIM1 has five inputs (BSRTDI, TRSTZ, TMS, TCK, and TDI) and seven outputs (DMX, BSRCLK, DRSHIFTZ, DRHOLDZ, RESETZ, TDO, and SDOENA). The boundary scan register TDI (BSRTDI) input is connected to the TDO output of the boundary scan register. The boundary scan register clock (BSRCLK) output is the clock input to the boundary scan register.

Figure 8-64. Example 1149.1 Test Architecture Using TIM1



The TIM1 receives input from the TRSTZ, TMS, TCK, and TDI test pins, and outputs control signals (DMX, BSRCLK, DRSHIFTZ, DRHOLDZ, RESETZ, TDO, and SDOENA) to the boundary scan register and TDO 3-state buffer.

The boundary scan register receives the TDI signal, the DMX, BSRCLK, DRSHIFTZ and DRHOLDZ control signals from TIM1, and data input from the IC core logic and input pins. The boundary scan register outputs a TDO signal to TIM1 and data signals to the IC core logic and output pins.

During instruction-register scan operations, TIM1 shifts data through its internal two-bit instruction register from the TDI input to the TDO output pin. During boundary scan register scan operations, TIM1 outputs control to shift data through the boundary scan register, from the TDI input pin to the BSRTDI input of TIM1, through TIM1 to the TDO output pin. During bypass-register scan operations, TIM1 shifts data through its internal BYPREG, from the TDI input to the TDO output pin.

During any scan operation, the data at the TDI pin is input during the rising edge of TCK and the data at the TDO pin is output on the falling edge of TCK. If the TIM1 TRSTZ input is low, the internal instruction register is initialized with a BYPASS instruction, the TDI, TMS, and TCK signals are ignored, and the TDO output buffer is 3-state.

Note:

Performing a data-register scan operation after the TIM1 has been initialized, by either activating the TRSTZ pin or by scanning the TAP into its test logic reset, will cause data to be shifted through the BYPREG from the TDI input to TDO output pin.

8.5.8.4 Module Count Calculation Using TIM1

When designing the 1149.1 boundary scan architecture into FPGAs, you must know the number of modules required by the test logic. The following equation lets you calculate the number of modules when using TIM1 (see Table 8-26 for module counts for each 1149.1 test cell):

Module Count = TIM1 + Boundary Register

Where:

TIM1 = 79 modules
Boundary register = number of modules in B cells

Assuming the boundary scan register consists of 40 UBCELL2 cells, calculate the module count:

Module Count = TIM1 + Boundary Register
= 79 + 40 • 4
= 239 Modules

8.5.8.5 Functional Description of TIM1

The function of TIM1 during data and instruction scan operations is summarized in the description statements shown in Table 8-27. TIM1 has a 2-bit instruction register and supports the EXTEST, SAMPLE/PRELOAD, INTEST, and BYPASS instructions. The 2-bit instruction decode for each instruction is shown in the description statements.

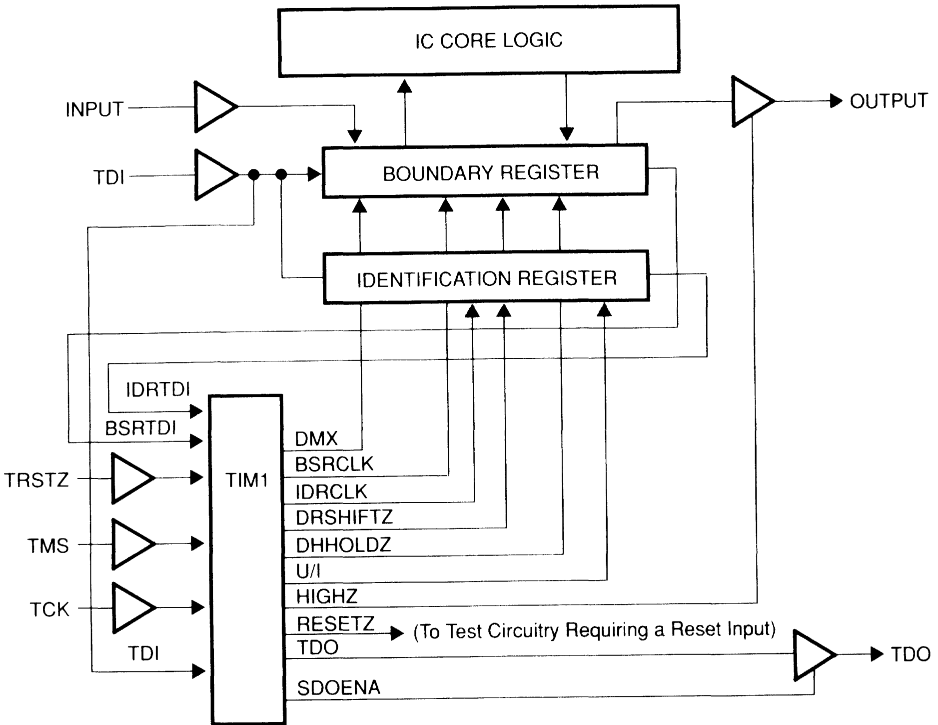
Table 8-27. TIM1 Functional Description

```
During Instruction Scan Operations:TDO-->IOUT1-->IOUT0-->TDO
                                   DMX=Previous Instruction Value
                                   BSRCLK=0
     When IOUT1:IOUT0=0:0, Instruction = EXTEST
     When IOUT1:IOUT0=0:1, Instruction = SAMPLE
     When IOUT1:IOUT0=1:0, Instruction = INTEST
     When IOUT1:IOUT0=1:1, Instruction = BYPASS
During EXTEST Data Scan Operations: BSRTDO-->TDO
                                    DMX=1
                                    BSRCLK=DRCK
During SAMPLE Data Scan Operations: BSRTDO-->TDO
                                    DMX=0
                                    BSRCLK=DRCK
During INTEST Data Scan Operations: BSRTDO-->TDO
                                    DMX=1
                                    BSRCLK=DRCK
During BYPASS Data Scan Operations: TDI-->BYPREG-->TDO
                                    DMX=0
                                    BSRCLK=0
```

8.5.8.6 Designing 1149.1 Using a TIM2 Circuit

The test interface macro 2 (TIM2) shown in Figure 8-65 is designed to simplify the construction of 1149.1 test structures in FPGA devices. TIM2 is a predefined 1149.1 interface consisting of a TAP, a three-bit instruction register, a bypass register, and a TDOCELL.

Figure 8-65. Example 1149.1 Test Architecture Using TIM2



TIM2 includes all the required test instructions of TIM1, plus additional instructions supporting the optional 1149.1 instructions: HIGHZ, CLAMP, IDCODE, and USERCODE. When TIM2 is used to construct an 1149.1 test structure, you must design only the boundary scan register, select either IDREG1 or IDREG2, and connect the registers to TIM2. TIM2 has six inputs: IDRTDI, BSRTDI, TRSTZ, TMS, TCK, and TDI; and ten outputs: DMX, BSRCLK, IDRCLK, DRSHIFTZ, DRHOLDZ, U/I (user/ID code), HIGHZ, RESETZ, TDO, and SDOENA.

The boundary scan register TDI (BSRTDI) input is connected to the TDO output of the boundary scan register. The boundary scan register clock (BSRCLK) output is the clock input to the boundary scan register. The identification register TDI (IDRTDI) input is connected to the TDO output of the selected identification register macro, IDREG1 or IDREG2. The identification register clock (IDRCLK) output is the clock input to the selected identification register, IDREG1 or IDREG2.

An example implementation of an 1149.1 boundary scan architecture using the TIM2 library component is shown in Figure 8-65.

The test architecture consists of the predefined TIM2 component. In addition, you define the boundary scan register section, which is composed of boundary scan cells (UBCELL1, UBCELL2, BBCELL1, and BBCELL2) that you select along with the identification register section (IDREG1 or IDREG2).

TIM2 contains the required 1149.1 EXTEST, BYPASS, and SAMPLE/PRELOAD instructions, as well as the optional INTEST, CLAMP, HIGHZ, IDCODE, and USERCODE instructions.

TIM2 receives input from the TRSTZ, TMS, TCK, and TDI test pins and outputs control signals DMX, BSRCLK, IDRCLK, DRSHIFTZ, DRHOLDZ, U/I, HIGHZ, RESETZ, TDO, and SDOENA to the boundary scan register, identification register, and TDO 3-state buffer.

The boundary scan register receives the TDI test pin signal; the TIM2 control signals DMX, BSRCLK, DRSHIFTZ, and DRHOLDZ; and data input from the IC core logic and input pins. The boundary scan register outputs a TDO signal to TIM2 and data signals to the IC core logic and output pins.

The identification register receives the TDI test pin signal and the IDRCLK, DRSHIFTZ, and U/I control signals from TIM2. The identification register outputs a TDO signal to TIM2.

During instruction register scan operations, TIM2 shifts data through its internal 3-bit instruction register from the TDI input to the TDO output pin.

During boundary scan register scan operations, TIM2 outputs control to shift data through the boundary scan register, from the TDI input pin to the BSRTDI input of TIM2, through TIM2 to the TDO output pin.

During identification register scan operations, TIM2 outputs control to shift data through the identification register, from the TDI input pin to the IDRTDI input of TIM2, through TIM2 to the TDO output pin.

During bypass register scan operations, TIM2 shifts data through its internal BYPREG, from the TDI input to the TDO output pin.

During any scan operation, the data at the TDI pin is input during the rising edge of TCK and the data at the TDO pin is output on the falling edge of TCK. If the TIM2 TRSTZ input is low, the internal instruction register is initialized with an IDCODE instruction, the TDI, TMS, and TCK signals are ignored, and the TDO output buffer is 3-state.

Note:

Performing a data-register scan operation after the TIM2 is initialized, either by activating the TRSTZ pin or scanning the TAP into its test logic reset, causes data to be shifted through the identification register from the TDI input to TDO output pin.

Note:

If the identification register in the TIM2 architecture is not used (see Figure 8-65), a BYPREG cell must be wired in place of the identification register. The BYPREG cell should be wired to receive the TDI input and the IDRCLK and DRSHIFTZ signals from TIM2 and to output a TDO signal to the IDRTDI input of TIM2. The BYPREG completes the serial data connection between the TDI and TDO pins during IDCODE and USERCODE data scan operations.

8.5.8.7 Module Count Calculation Using TIM2

When designing the 1149.1 boundary scan architecture into FPGAs, you must know the number of modules required by the test logic. The following equation lets you calculate the number of modules used using the TIM2 cell (see Table 8-26 for module counts for each 1149.1 test cell):

Module Count = TIM2 + IDREG + Boundary Register

Where:

TIM2 = 90 modules
IDREG1 = 68 modules
IDREG2 = 102 modules
Boundary register = Number of modules in B cells

Assuming a design with an IDREG1 identification register and a boundary scan register with 40 UBCELL2 cells, calculate the module count as shown:

Module Count = TIM2 + IDREG1 + Boundary Register
= 90 + 68 + (40 • 4)
= 318 Modules

A BYPREG cell can be substituted for the identification register section of the equation. The BYPREG cell has a module count of 2. Calculate the module count using the BYPREG cell instead of the IDREG1 cells as shown:

Module Count = TIM2 + BYPREG + Boundary Register
= 90 + 2 + 40 • 4
= 252 Modules

8.5.8.8 Functional Description of TIM2

Table 8-28 summarizes the TIM2 function during data and instruction scan operations. TIM2 has a 3-bit instruction register and supports the EXTEST, SAMPLE/PRELOAD, INTEST, CLAMP, HIGHZ, IDCODE, USERCODE, and BYPASS instructions. The 3-bit instruction decode for each instruction is shown in Table 8-28.

Table 8-28. TIM2 Functional Description

```
During Instruction Scan Operations:TDI-->OUT2-->OUT1-->OUT0-->TDO
                                   DMX, HIGHZ, U/I= Previous
                                   instruction value
                                   BSRCLK and IDRCLK=0
  When IOUT2:OUT1:OUT0=0:0:0, Instruction = EXTEST
  When IOUT2:OUT1:OUT0=0:0:1, Instruction = SAMPLE
  When IOUT2:OUT1:OUT0=0:1:0, Instruction = INTEST
  When IOUT2:OUT1:OUT0=0:1:1, Instruction = CLAMP
  When IOUT2:OUT1:OUT0=1:0:0, Instruction = HIGHZ
  When IOUT2:OUT1:OUT0=1:0:1, Instruction = IDCODE
  When IOUT2:OUT1:OUT0=1:1:0, Instruction = USERCODE
  When IOUT2:OUT1:OUT0=1:1:1, Instruction = BYPASS
During EXTEST Data Scan Operations: BSRTDI-->TDO
                                   DMX=1, HIGHZ=1, U/I=0
                                   BSRCLK=DRCK, IDRCLK=0
During SAMPLE Data Scan Operations: BSRTDI-->TDO
                                   DMX=0, HIGHZ=1, U/I=0
                                   BSRCLK=DRCK, IDRCLK=0
During INTEST Data Scan Operations: BSRTDI-->TDO
                                   DMX=1, HIGHZ=1, U/I=0
                                   BSRCLK=DRCK, IDRCLK=0
During CLAMP Data Scan Operations: TDI-->BYPASS-->TDO
                                   DMX=1, HIGHZ=1, U/I=0
                                   BSRCLK and IDRCLK=0
During HIGHZ Data Scan Operations: TDI-->BYPASS-->TDO
                                   DMX=1, HIGHZ=0, U/I=0
                                   BSRCLK and IDRCLK=0
During IDCODE Data Scan Operations: IDRTDI-->TDO
                                   DMX=0, HIGHZ=1, U/I=0
                                   BSRCLK=0, IDRCLK=DRCK
During USERCODE Data Scan Operations: IDRTDI-->TDO
                                   DMX=0, HIGHZ=1, U/I=1
                                   BSRCLK=0, IDRCLK=DRCK
During BYPASS Data Scan Operations: TDI-->BYPASS-->TDO
                                   DMX=0, HIGHZ=1, U/I=0
                                   BSRCLK and IDRCLK=0
```

8.6 Pipelined Multiplier†

This section describes the design of a pipelined multiplier circuit implemented in FPGA technology with an emphasis on optimizing the circuit for speed. The architecture is developed directly from an examination of the binary multiplication process. Speed is obtained at the expense of die resource consumption. Final performance is approximately 11 Mhz at 70°C and a supply voltage of 4.75 V for a 10-bit multiplier and multiplicand. A mix of capture methods are used, including schematic capture and Boolean equation entry. The register and routing rich construction of the TPC12 series TI FPGA proved to be advantageous for this implementation, specifically the TPC1280.

A commonly desired function for FPGA implementation is multiplication and one of the most desirable qualities of a multiplier is speed. Selecting an appropriate architecture is an interesting and challenging problem, particularly when constrained to a given set of circuit building blocks as in FPL implementations.

Many possible architectures exist, including serial sequential and parallel combinational schemes; however, the topology yielding the best solution for FPL is a pipelined topology. Multiplication consists of a discrete number of functions that can be executed independently and combined to obtain the final product. Furthermore, these independent steps can be executed in series for a number of operand pairs with the *n*th pair relinquishing access to each stage to the *n+1* pair; i. e., each pair of operands proceeds through the circuitry and a function is performed at each step much like a manufacturing assembly line.

8.6.1 Mathematical Basis

The assembly line processing described above is an obvious choice when the process of binary multiplication is analyzed. An example multiplication is shown in Table 8-17, with *A(9:0)* representing the multiplicand and *B(9:0)* representing the multiplier. The partial products are given by *C(9:0)* through *L(9:0)*. The result is given by *Z(J:0)*.

Table 8-29. Example Multiplication

										A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
									X	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
										C9	C8	C7	C6	C5	C4	C3	C2	C1	C0
									D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	
								E9	E8	E7	E6	E5	E4	E3	E2	E1	E0		
							⋮			⋮				⋮					
	L9	L8	L7	L6	L5	L4	L3	L2	L1	L0				⋮					
ZJ	ZI	ZH	ZG	ZF	ZE	ZD	ZC	ZB	ZA	Z9	Z8	Z7	Z6	Z5	Z4	Z3	Z2	Z1	Z0

† Contributed by Joel S. Lason, P. E., FPGA Applications, Texas Instruments Incorporated.

Examination of this function reveals that partial products can be added in pairs and summed with succeeding partial products. Observe that the *D(9:0)* partial product is shifted left one digit and that each subsequent partial product is shifted two, three, and up to nine places. Each partial product is formed by multiplying *A(9:0)* by one bit of the multiplier. For example, *C(9:0)* is formed by multiplying *A(9:0)* by *B0*, *D(9:0)* by *B1*, etc.

The least-significant bit of each multiplier participates in only one addition (although *C0* can be passed directly to the product output without undergoing an addition). The remaining additions of the partial products pass the least-significant bit of the first operand without allowing an input to the adder, allowing 10-bit adders to be used throughout the circuit without having to resort to larger adders.

Pipelined multiplication can be performed by generating a partial product, shifting it left in relation to the previous partial sum, and summing the partial product and the previous sum. This function can be repeated with registers to separate the stages. The registers allow the next pair of operands to enter the pipeline even though the previous pair has not completed the operation.

By breaking up the process this way each accumulating sum can be generated in a much shorter time than it would take for the operand pair to traverse the entire multiply-add chain from input to output. Once the pipeline has been filled, a product is generated on every clock cycle.

8.6.2 Other Methods Investigated

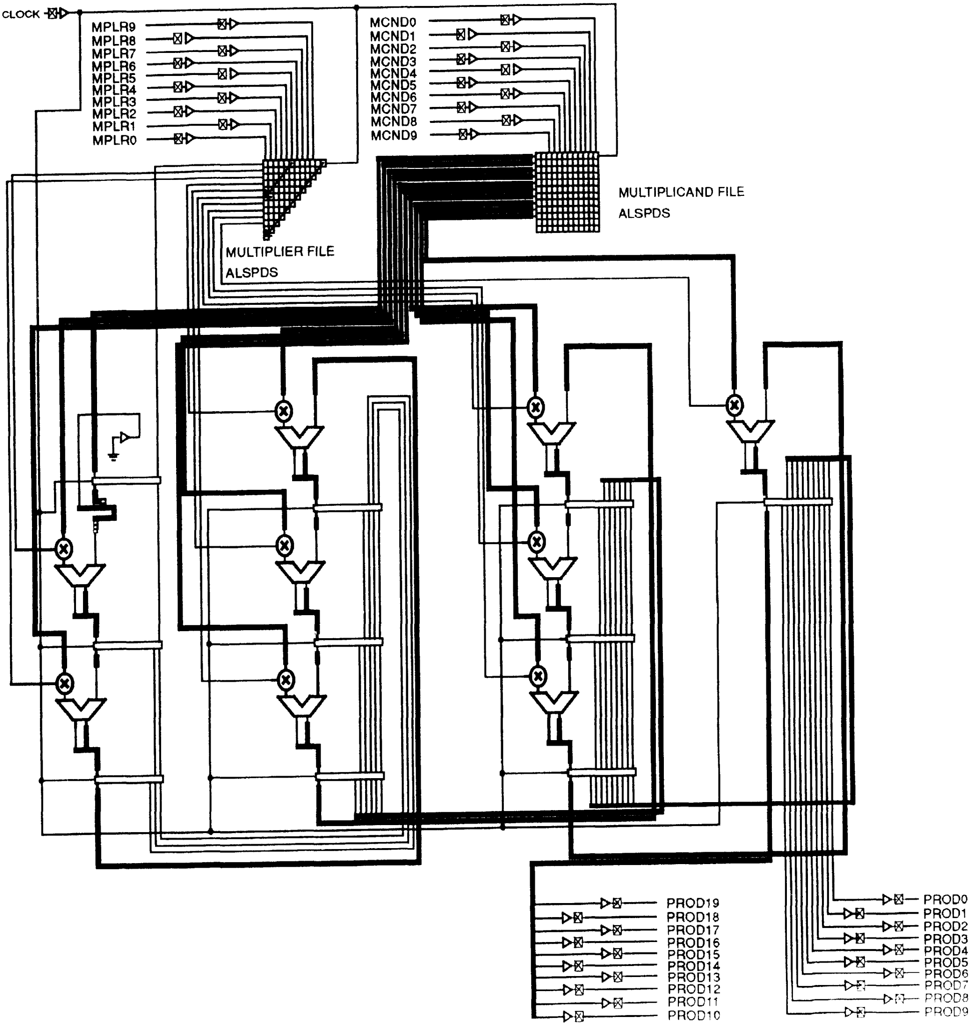
Although other architectures, such as serial, serial/parallel, and parallel solutions consume far fewer die resources, none are as fast as the pipelined solution. The serial solutions provide a product after a multiple of clock cycles and the parallel multiplier, which is a combinational circuit, introduces long paths through the circuit. The signals must ripple through these long paths before a result can be obtained.

After the pipeline is started a product is obtained every clock cycle, which is not possible with other solutions, although It takes several periods for the first product to be completely evaluated, with the number of periods a function of the word width, but thereafter a product is available every clock cycle. Each product is delayed by a number of cycles equal to the word width of the multiplier. This relationship between the inputs and outputs is more obvious when a timing diagram of the circuit is analyzed in the section on simulation.

8.6.3 Design Capture

The design is captured in both schematic and boolean equation form. The top-level schematic is captured in a way that indicates the data flow and architecture (Figure 8-66).

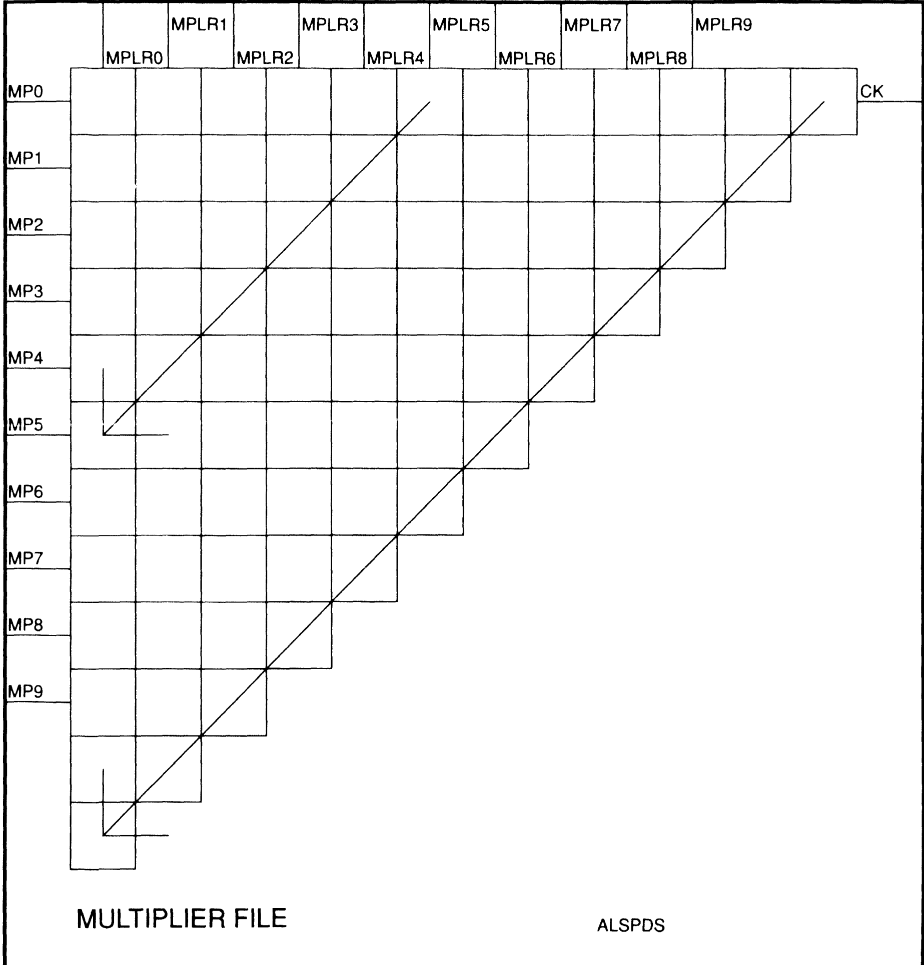
Figure 8-66. Top-Level Schematic



The schematic contains four basic types of components: multiplier file, multiplicand file, multiplier/adder, and registers of various widths. A sublevel schematic describes the multiplier/adder; the other blocks are represented by Boolean equation text files. The schematic symbols represent a graphical description of the function of each part.

The multiplier file is located at the upper left of the top-level schematic. An enlargement of the symbol is shown in Figure 8-67.

Figure 8-67. Multiplier File Schematic Symbol



The 10 bits of the multiplier are clocked into the 10 inputs on top of the symbol once each rising edge of the clock. At each succeeding rising edge of the clock another 10-bit multiplier is clocked into the multiplier file and the previous multiplier is transferred to the next stage in the file and shifted right, which is the same as discarding the least-significant bit of the multiplier each time a new multiplier is clocked into the device. Because each multiplier bit is only needed for one operation to generate a partial product, the least-significant bit can be discarded at each transfer.

Each output of the multiplier file is connected to the multiplier input of the multiplier/adder block and routed to one of the 10 pipeline stages. This block is described by Boolean equations written in PALASM syntax (Figure 8-68).

Figure 8-68. Multiplier File Block Description

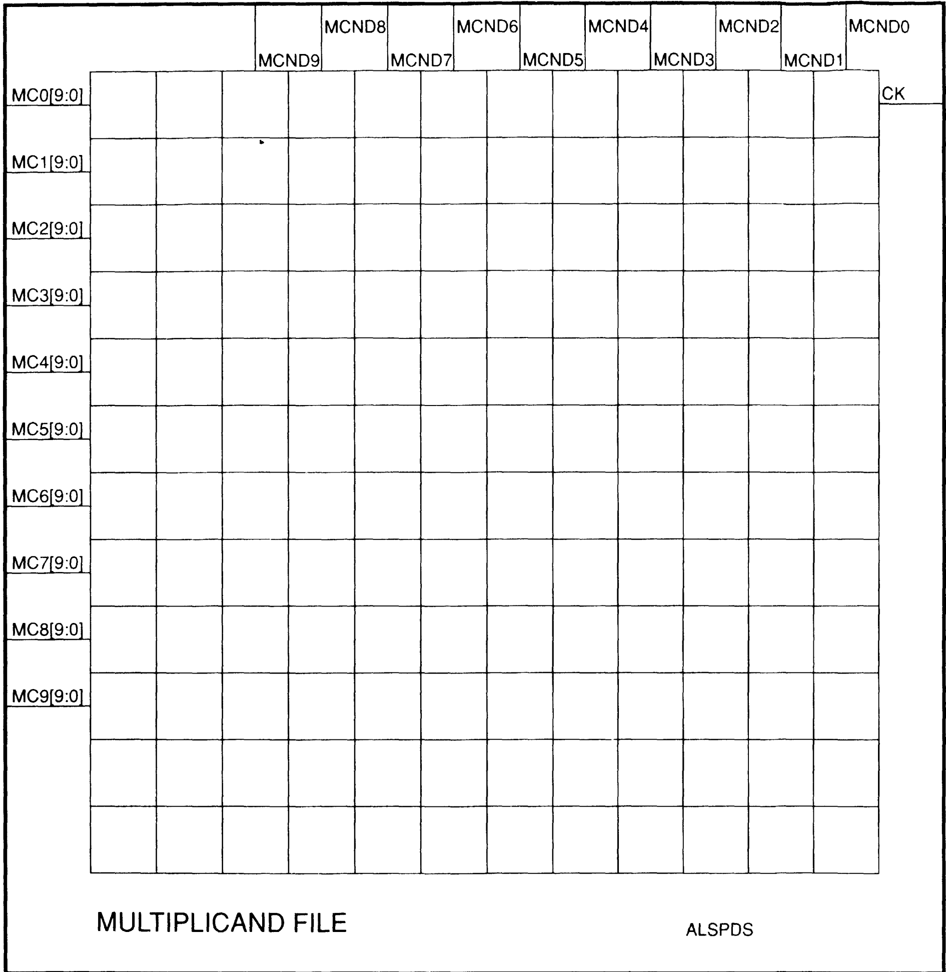
```
COMPANY TEXAS INSTRUMENTS INC.
AUTHOR JOEL S. LASON
DATE SEPTEMBER 7 1992
CHIP MPLRFILE USER
CK MPO MP1 MP2 MP3 MP4 MP5 MP6 MP7 MP8 MP9
MPLRO MPLR1 MPLR2 MPLR3 MPLR4 MPLR5 MPLR6 MPLR7 MPLR8 MPLR9
EQUATIONS
MP0:=MPLR0;
MP01:=MPLR1;
MP02:=MPLR2;
MP03:=MPLR3;
MP04:=MPLR4;
MP05:=MPLR5;
MP06:=MPLR6;
MP07:=MPLR7;
MP08:=MPLR8;
MP09:=MPLR9;
MP1:=MP01;
MP11:=MP02;
MP12:=MP03;
MP13:=MP04;
MP14:=MP05;
MP15:=MP06;
MP16:=MP07;
MP17:=MP08;
MP18:=MP09;
MP2:=MP11;
MP21:=MP12;
MP22:=MP13;
MP23:=MP14;
MP24:=MP15;
MP25:=MP16;
MP26:=MP17;
MP27:=MP18;
```

```
MP3:=MP21;
MP31:=MP22;
MP32:=MP23;
MP33:=MP24;
MP34:=MP25;
MP35:=MP26;
MP36:=MP27;
MP4:=MP31;
MP41:=MP32;
MP42:=MP33;
MP43:=MP34;
MP44:=MP35;
MP45:=MP36;
MP5:=MP41;
MP51:=MP42;
MP52:=MP43;
MP53:=MP44;
MP54:=MP45;
MP6:=MP51;
MP61:=MP52;
MP62:=MP53;
MP63:=MP54;
MP7:=MP61;
MP71:=MP62;
MP72:=MP63;
MP8:=MP71;
MP81:=MP72;
MP9:=MP81;
```

Conversion from PALASM to a format that can be simulated is performed by TI-ALES, which also generates a netlist that is read by the back-end tools performing rule checking, placement, and routing. The simulator understands that the multiplier file is described this way because the special attribute *ALSPDS* is attached to the symbol.

The multiplicand file symbol, shown in Figure 8-69, stores incoming multiplicands for 10 clock cycles and transfers each through the file to be used by each stage in turn.

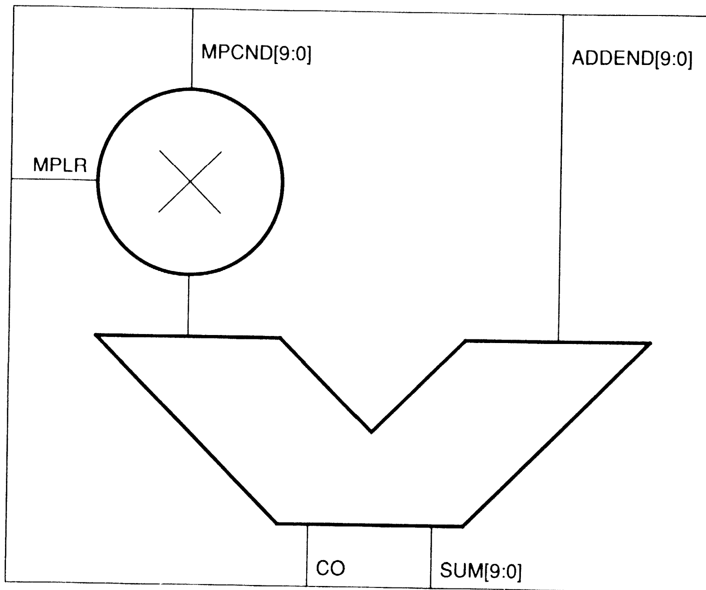
Figure 8-69. Multiplicand File Schematic Symbol



The multiplicands enter through the top of the file and exit at the bus pins on the side of the file. Each bus pin is routed to a bus input on the multiplier/adder block. The multiplicand file is a rectangular array of registers, each of which transfers data to the input of the next register and makes its output available to the multiplier/adder.

The multiplier/adder block is the only block described by a sublevel schematic (Figure 8-70).

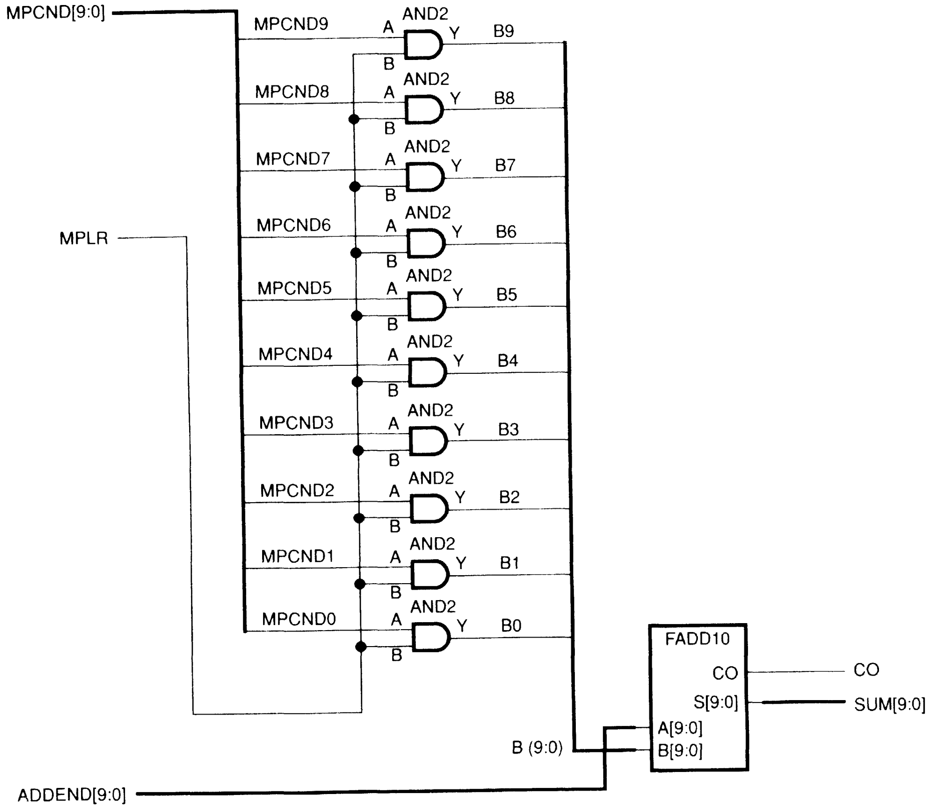
Figure 8-70. Multiplier/Adder Block Schematic Symbol



The symbol indicates that a partial product is generated from one of the multiplier bits sourced from the multiplier file and the multiplicand that is sourced from the multiplicand file. This partial product is then summed with the preceding partial sum that is generated by summing all the partial products generated prior to this stage.

The sublevel multiplier/adder schematic shows how the multiply/add is performed (Figure 8-71).

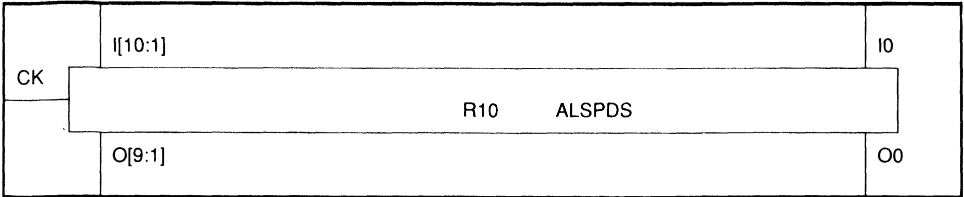
Figure 8-71. Sublevel Multiplier/Adder Schematic



Ten AND gates generate the partial product. The FADD10 macro, a standard TI FPGA library part, is optimized to perform fast addition.

The register, the last major building block, is actually a class of blocks that perform the same function but have different bit widths depending on their place in the pipeline (Figure 8-72).

Figure 8-72. Register Block Schematic Symbol



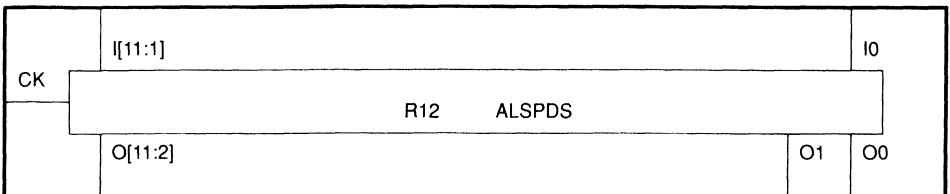
The first register, R10, differs from the others. Because a partial product has to be generated first and there is no accumulating sum, R10 equations perform multiplication and register functions. All other blocks only register the current state of the multiplication. The function of R10 is indicated by its PALASM source file (Figure 8-73).

Figure 8-73. Register Block PALASM Source File

```
COMPANY TEXAS INSTRUMENTS INC.
AUTHOR JOEL S. LASON
DATE SEPTEMBER 7 1992
CHIP R10 USER
CK I10 I9 I8 I7 I6 I5 I4 I3 I2 I1 I0
O9 O8 O7 O6 O5 O4 O3 O2 O1 O0
EQUATIONS
O9:=I10*I0;
O8:=I9*I0;
O7:=I8*I0;
O6:=I7*I0;
O5:=I6*I0;
O4:=I5*I0;
O3:=I4*I0;
O2:=I3*I0;
O1:=I2*I0;
O0:=I1*I0;
```

A typical register configuration is shown by the R12 block (Figure 8-74).

Figure 8-74. R12 Block Schematic Symbol



The R12 block stores the state of the calculation and performs the necessary shift of the partial product by taking the least-significant bit of the sum and passing it on to the next stage's register because it no longer participates in the calculation (see Section 8.6.1).

The other components on the top-level schematic are the input, output, and clock buffers, all standard library parts. The data flow through the circuit follows from the definition of its component blocks. The multiplier and multiplicand are stored by their respective files and applied to the first stage. The partial product generation is performed and registered on the next clock. This clock also stores new multipliers and multiplicands. After the second clock, the data ripples through the next stage, which involves generation of the next partial product and adding it to the previous partial product.

The process continues until the calculation has proceeded through all 10 stages generating, summing, and shifting all the partial products.

8.6.4 Logic Simulation

The results of logic simulation are shown in waveform format in Figure 8-75 and Figure 8-76.

Figure 8-75. Logic Simulation Waveform (1 of 2)

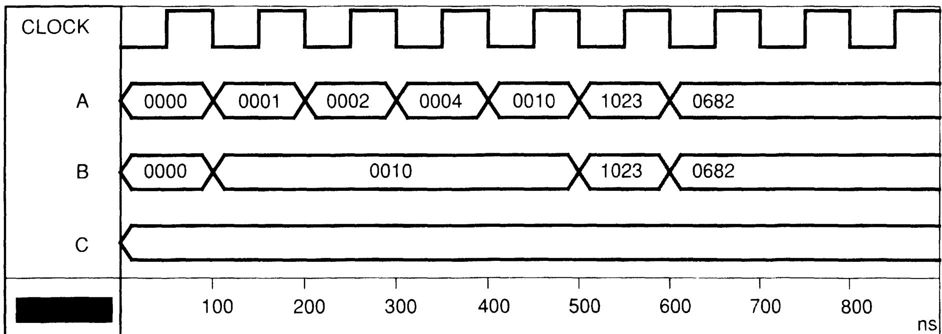
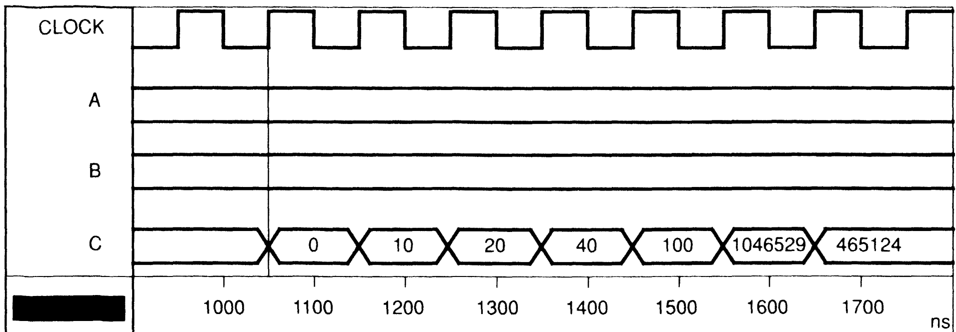


Figure 8-76. Logic Simulation Waveform (2 of 2)



The master clock is given a period of 100 ns. Seven multiplier/multiplicand pairs are provided to the circuit and clocked through providing correct results as indicated on the waveforms and in the text output from the simulation log file shown in Figure 8-77.

Figure 8-77. Simulation Log File

```
time =  100.0ns CLOCK=1 A=0000\D B=0000\D C=XXXXXXX\D
time =  200.0ns CLOCK=1 A=0001\D B=0010\D C=XXXXXXX\D
time =  300.0ns CLOCK=1 A=0002\D B=0010\D C=XXXXXXX\D
time =  400.0ns CLOCK=1 A=0004\D B=0010\D C=XXXXXXX\D
time =  500.0ns CLOCK=1 A=0010\D B=0010\D C=XXXXXXX\D
time =  600.0ns CLOCK=1 A=1023\D B=1023\D C=XXXXXXX\D
time =  700.0ns CLOCK=1 A=0682\D B=0682\D C=XXXXXXX\D
time =  800.0ns CLOCK=1 A=0682\D B=0682\D C=XXXXXXX\D
time =  900.0ns CLOCK=1 A=0682\D B=0682\D C=XXXXXXX\D
time = 1000.0ns CLOCK=1 A=0682\D B=0682\D C=XXXXXXX\D
time = 1100.0ns CLOCK=1 A=0682\D B=0682\D C=0000000\D
time = 1200.0ns CLOCK=1 A=0682\D B=0682\D C=0000010\D
time = 1300.0ns CLOCK=1 A=0682\D B=0682\D C=0000020\D
time = 1400.0ns CLOCK=1 A=0682\D B=0682\D C=0000040\D
time = 1500.0ns CLOCK=1 A=0682\D B=0682\D C=0000100\D
time = 1600.0ns CLOCK=1 A=0682\D B=0682\D C=1046529\D
time = 1700.0ns CLOCK=1 A=0682\D B=0682\D C=0465124\D
time = 1800.0ns CLOCK=1 A=0682\D B=0682\D C=0465124\D
Simulation stopped at 1800.0ns
```

Note:

The simulation uses unit delays and does not reflect routing and loading delays.

The first operands are applied to the circuit at 50 ns. The result appears at the output of the circuit at 1050 ns, or 10 clock cycles later. Results of the following operand pair appear at the output one cycle later. The circuit does not require another 10 cycles to generate subsequent products as long as the operands enter the pipeline immediately following the first pair.

8.6.5 Static Timing Analysis

Static timing analysis is performed from within the TI-ALS tool set using the timer program. The timer tool provides information about circuit performance without requiring the circuit stimulus that a timing simulation requires. Static timing analysis is performed by specifying signals or groups of signals as starting and ending points between which circuit delays are measured. Internal frequency, clock-to-output delay, and external setup time are specific measurements that make sense for this design.

The internal frequency measurement determines how fast the programmed device can operate internally without considering effects of loads on the device pins. For a synchronous sequential circuit like this 10-bit multiplier, the internal speed is given by the register-to-register delay that includes the combinational logic between register stages. To obtain this information, commands are issued requesting the time along a path from the clock input of the first component to the next clocked input down the path. This circuit produces the results shown in Table 8-30.

Table 8-30. Internal Frequency Measurement

1st longest path to all endpins

Rank	Total	Start Pin	First Net	End Net	End Pin
0	81.8	$112/I50:CLK	MP8	CO8	$11496/I13:D
1	81.6	$112/I50:CLK	MP8	SUM88	$11496/I7:D
2	81.6	$112/I50:CLK	MP8	SUM89	$11496/I0:D
3	81.5	$112/I22:CLK	MP7	SUM78	$11495/I14:D
4	78.9	$112/I61:CLK	MP9	SUM98	$11497/I6:D
5	77.7	$112/I1:CLK	MP6	CO6	$11494/I0:D
6	75.6	$112/I22:CLK	MP7	CO7	$11495/I15:D
7	75.5	$112/I61:CLK	MP9	CO9	$11497/I9:D
8	74.9	$112/I1:CLK	MP6	SUM68	$11494/I3:D
9	74.9	$112/I22:CLK	MP7	SUM79	$11495/I6:D

--- 299 more items ---

The table shows that the longest path from a clocked input through a string of combinational logic and up to the D input of a D flip-flop at the end of the path is 81.8 nanoseconds. This corresponds to a system frequency of 12.2 Mhz. The operating conditions supplied to the timer program are 70°C and 4.75 V V_{CC}. Different conditions can be specified, if desired.

Another important performance parameter is clock-to-output delay, which is the delay from the clocking of an internal sequential component to the appearance of valid data at the output pin of the device. A standard pin loading is assumed. The results of this analysis are shown in Table 8-31.

Table 8-31. Clock-to-Output Delay

1st longest path to all endpins

Rank	Total	Start Pin	First Net	End Net	End Pin
0	40.6	CLKBUF/U0:PAD	$1N187	PROD12	PR_BUF12:PAD
1	39.9	CLKBUF/U0:PAD	$1N187	PROD11	PR_BUF11:PAD
2	39.8	CLKBUF/U0:PAD	$1N187	PROD15	PR_BUF15:PAD
3	37.0	CLKBUF/U0:PAD	$1N187	PROD14	PR_BUF14:PAD
4	35.2	CLKBUF/U0:PAD	$1N187	PROD13	PR_BUF13:PAD
5	34.9	CLKBUF/U0:PAD	$1N187	PROD16	PR_BUF16:PAD
6	34.8	CLKBUF/U0:PAD	$1N187	PROD10	PR_BUF10:PAD
7	34.3	CLKBUF/U0:PAD	$1N187	PROD7	PR_BUF7:PAD
8	32.5	CLKBUF/U0:PAD	$1N187	PROD18	PR_BUF18:PAD
9	32.5	CLKBUF/U0:PAD	$1N187	PROD8	PR_BUF8:PAD

--- 11 more items ---

This path operates more quickly than the internal clock-to-data paths and is not the limiting factor in system performance.

To determine the external setup time requirement, the delay must be determined from the data input pads to their respective registers t(datapath) and from the clock pads to the clock inputs t(clock). From this data the equation for setup time can be evaluated according to the formula:

$t(ext) > t(datapath) - t(clock)$

The data shown in Table 8-32 indicates that the largest value for t(datapath) is 22.3 ns. Only the first 10 values for t(clock) are shown in Table 8-33.

Table 8-32. t(datapath) Delays

1st longest path to all endpins

Rank	Total	Start Pin	First Net	End Net	End Pin
0	22.3	MP_BUF1:PAD	MP_IN1	$1I2/N69	$1I2/I54:D
1	19.4	MP_BUF0:PAD	MP_IN0	MP_IN0	$1I2/I21:D
2	19.3	MC_BUF3:PAD	MCND_IN3	MCND_IN3	$1I3/I10:D
3	17.0	MC_BUF2:PAD	MCND_IN2	MCND_IN2	$1I3/I98:D
4	15.9	MC_BUF6:PAD	MCND_IN6	MCND_IN6	$1I3/I4:D
5	15.9	MP_BUF3:PAD	MP_IN3	$1I2/N74	$1I2/I45:D
6	15.8	MC_BUF5:PAD	MCND_IN5	MCND_IN5	$1I3/I5:D
7	15.8	MC_BUF1:PAD	MCND_IN1	MCND_IN1	$1I3/I79:D
8	15.7	MP_BUF8:PAD	MP_IN8	$1I2/N67	$1I2/I46:D
9	15.3	MP_BUF2:PAD	MP_IN2	$1I2/N68	$1I2/I0:D
10	15.2	MP_BUF9:PAD	MP_IN9	$1I2/N73	$1I2/I44:D
11	14.6	MP_BUF7:PAD	MP_IN7	$1I2/N71	$1I2/I11:D
12	14.6	MC_BUF9:PAD	MCND_IN9	MCND_IN9	$1I3/I30:D
13	14.6	MC_BUF7:PAD	MCND_IN7	MCND_IN7	$1I3/I23:D
14	14.2	MP_BUF6:PAD	MP_IN6	$1I2/N66	$1I2/I28:D
15	14.1	MC_BUF8:PAD	MCND_IN8	MCND_IN8	$1I3/I81:D
16	14.1	MC_BUF0:PAD	MCND_IN0	MCND_IN0	$1I3/I14:D
17	14.1	MP_BUF5:PAD	MP_IN5	$1I2/N72	$1I2/I52:D
18	13.8	MC_BUF4:PAD	MCND_IN4	MCND_IN4	$1I3/I11:D
19	13.5	MP_BUF4:PAD	MP_IN4	$1I2/N70	$1I2/I26:D

--- 289 more items ---

Table 8-33. t(clock) Delays

1st longest path to all endpins

Rank	Total	Start Pin	First Net	End Net	End Pin
0	16.8	CLKBUF/U0:PAD	$1N187	$1N187	$1I489/I8:CLK
1	16.8	CLKBUF/U0:PAD	$1N187	$1N187	$1I3/I21:CLK
2	16.8	CLKBUF/U0:PAD	$1N187	$1N187	$1I3/I65:CLK
3	16.8	CLKBUF/U0:PAD	$1N187	$1N187	$1I489/I1:CLK
4	16.8	CLKBUF/U0:PAD	$1N187	$1N187	$1I490/I13:CLK
5	16.8	CLKBUF/U0:PAD	$1N187	$1N187	$1I3/I78:CLK
6	16.8	CLKBUF/U0:PAD	$1N187	$1N187	$1I493/I15:CLK
7	16.8	CLKBUF/U0:PAD	$1N187	$1N187	$1I490/I4:CLK
8	16.8	CLKBUF/U0:PAD	$1N187	$1N187	$1I3/I5:CLK
9	16.8	CLKBUF/U0:PAD	$1N187	$1N187	$1I3/I20:CLK

--- 299 more items ---

The value for t(clock) that goes to the same register is Rank 71 is not shown but is16.2 ns. Using these numbers, an external setup time of 6.1 ns is derived. The value of t(clock) with the shortest delay (14.7 ns) and the value of t(datapath) with the longest delay (22.3 ns) gives a worst-case value of 7.6 ns at 4.75 V V_{CC} and 70°C.

The TI FPGA TPC12 series architecture is a natural fit for dense, register-intensive functions. The TPC12 series device has two types of logic modules: combinational and combinational/sequential. The combinational/sequential device provides the advantage for this type of circuit. For a description of the TPC12 series architecture, see Section 1.2.4.

Between each register of the pipeline is a block of combinational logic. Because the TPC12 series combinational/sequential module has a combinational section first, some combinational adder circuitry is absorbed into the module, which eliminates one level of logic and improves design performance.

In addition, the small granularity of both types of logic modules offers another advantage. Granularity refers to the relative size of the logic module compared to other potential module implementations. If a large implementation is chosen it can eliminate levels of logic but it also prevents packing as much circuitry into the device. For example, if an inverter needs to be implemented and there is no other logic that can be combined with it, most of a larger module will be wasted. This leads to low device utilization.

These advantages are realized with the TI-ALS development system placement and routing software. The combiner feature of the software combines the combinational and sequential functions into a single module. For the example design, 72 logic modules were removed. The final module usage for combinational modules is thus reduced from 923 to 851. The number of sequential modules used in the design is 309.

The software provided 100-percent automatic placement and routing of the example design in under an hour. Considering that the 1280 device has 1232 modules, this design used 94 percent of the device and automatically routed to completion in less than an hour.

For specific device, software, or applications questions, or to request a copy of the design database, please call Texas Instruments Field Programmable Gate Array Applications at (214)997-5666.

Glossary

These symbols, terms, and definitions are in accordance with those currently agreed upon by the JEDEC Council of the Electronic Industries Association (EIA) for use in the USA and by the International Electrotechnical Commission (IEC) for international use.

Part 1 – General Concepts

ADL Design language. The format for TI-ALS design data.

ALS Action Logic System

Action Logic System The development system for configuring, programming, and debugging TI FPGAs.

Actionprobe TI-ALS feature that allows internal circuit nodes to be observed at the external device pin.

Activate TI-ALS program that programs the TPC device.

Activator 1 Programming hardware for TPC10 series devices.

Activator 2 Programming hardware for TPC10 and TPC12 series devices.

Antifuse A term for the type of programming element used in TPC arrays. An antifuse is a fuse which is normally open, but converts to a resistive connection when programmed.

Array The area occupied by the rows of modules and the routing channels.

Assertion System error causing abnormal program termination. A failure code is written to the `\alsuser\user\assert.log` file. Errors of this type should be reported to the TI Field Programmable Logic (FPL) helpline at (214) 997-5666.

Back Annotation The process of translating data generated from the TI-ALS system back to the CAE design environment. Postlayout delay information is back annotated to the CAE simulator.

Binning circuit Nonuser circuit used to characterize the AC performance of a TPC device. All TPC devices contain a binning circuit that has been tested to an AC test limit to ensure that data sheet specifications have been met.

CAPTUREDR A Capture Data Register state in the TAP diagram allowing the selected data register to parallel load with test data.

CAPTUREIR A Capture Instruction Register state in the TAP diagram allowing the instruction register to parallel load with test status.

Channel The area reserved for routing; contains horizontal tracks and horizontal pass transistors.

Clock Network A system for distributing a global clock signal throughout the array, with minimum skew and large drive capability.

Configure — Process for determining placement and routing for a design.

Critical Net — Net whose signal propagation delay is part of a critical path in the design.

Criticality — Design property. The level of criticality assigned to a net influences the placement and routing of a net and may influence the propagation delay associated with the net. The four levels of criticality are fast critical (F), medium critical (M), uncritical (U) and default critical.

Criticality File — Design constraint file for assigning criticality to nets.

DCLK Pin — Device pin. When MODE=0, DCLK is a user I/O, and when MODE=1, DCLK is used by the program/debug hardware to clock data into internal program/test registers.

Debug — TI-ALS program for debugging programmed TPC devices.

Error — A design problem that must be corrected before TI-ALS will proceed.

EXIT1DR — A primary Exit Data Register scan state in the TAP allowing entry into either the UPDATEDR or PAUSEDR states.

EXIT1IR — A primary Exit Instruction Register scan state in the TAP allowing entry into either the UPDATEIR or PAUSEIR states.

EXIT2DR — A secondary Exit Data Register scan state in the TAP allowing entry into either the UPDATEDR or SHIFTDR states.

EXIT2IR — A secondary Exit Instruction Register scan state in the TAP allowing entry into either the UPDATEIR or SHIFTIR states.

Extract — TI-ALS program that creates postlayout delay information.

Fixed macro — A hard macro that has a user-assigned location that is not altered by automatic placement.

Fusemap — Design file containing a list of antifuse addresses used by the programming hardware to program the device.

Fuser — TI-ALS program that generates the fusemap for the TPC device.

Gate Equivalent Circuit — A basic unit of measure of relative digital circuit complexity. The number of gate equivalent circuits is that number of individual logic gates that would have to be interconnected to perform the same function.

GND — Device pin. All pins labeled GND must be connected to circuit ground.

Hard macro Unit-level macros that have a fixed relative placement. Two-module hard macros are always placed side by side. All routing between two-module hard macros is predetermined. Hard macros are predefined by Texas Instruments.

Hierarchy Style of organizing a design where a level of a logic design is expressed in terms of lower-level subcircuits or blocks. Lower-level blocks may, in turn, be expressed in terms of lower-level blocks.

ICP In-circuit probe debug command used to assign the Actionprobes to signals internal to a TPC device operating in the end system environment.

I/O Module Module used to configure I/O macros, like INBUF, TRIBUFF, etc.

I/O Pin The connection to the package lead that is bonded to the device I/O buffer.

Instance CAE term. Each placement of a library element or hierarchical block into a design defines a unique instance of that element. The instance may have a default or assigned name that is independent of the library element function.

Instance Pins An instance pin is a macro pin for a given instance. For example, the A-input of an AND2A (2-input AND gate) with an instance name of U1 has an instance pin name of U1:A.

LSB Least Significant Bit. In both the instruction and data scan registers it is defined to be the bit nearest the TDO output.

Label CAE term. A label is a method of attaching names, text, etc. to nets or instances in a design.

Level A property attached to macro symbols indicating which category of macros they belong to. Choices are unit, hard, soft, and user.)

Logic Compaction Logic compression. Logic conversion from standard TTL or CMOS technology into TI FPGA technology which combines functions from multiple logic elements into one logic module.

Logic Module Module used to configure logic macros, like AND, OR, etc. The basic logic building block of the TI FPGA from which all logical functions are built.

Long Horizontal Track A horizontal routing track is used to interconnect two (or more) pins separated by a substantial horizontal distance on a net. Typically, a long horizontal track occupies more than a third of the columns of the array. Only a limited number of such tracks are available. If more tracks are needed, the placement will fail. Nets using long horizontal tracks may encounter some additional delay.

Long Vertical Track A vertical routing track is used to interconnect two (or more) pins separated by a substantial vertical distance on a net. Typically, a long vertical track occupies more than three to six rows of the array. Only a limited number of tracks are available. If more tracks are needed, the placement will fail. Nets using long vertical tracks may encounter a substantially longer delay than the median.

MSB Most Significant Bit. In both the instruction and data scan registers it is defined to be the bit nearest the TDI input.

Macro Pin Macro pins are connection points to macro functions. For example, the A-input of an AND2A (2-input AND gate) is a macro pin.

MODE Pin Device pin which places the device in normal mode (MODE=0) or test/program mode (MODE=1).

Module Synonym for the basic functional block in a TPC array. Modules are used to construct macros (See Logic Module).

Net A logic signal path between macros. A net can be implemented in one or more routing segments connected by two or more antifuses.

PAUSEDR A Pause Data Register scan state in the TAP enabling a data scan operation to be suspended.

PAUSEIR A Pause Instruction Register scan state in the TAP enabling an instruction scan operation to be suspended.

Programmable Array Logic (PAL)
PAL circuits are user- programmable integrated circuits which utilize fuse link technology to implement logic functions. Implements sum of products logic by using a programmable AND array whose outputs feed a fixed OR array.

Programmable Logic Device (PLD)
User-programmable device (see above)

PLICE (Programmable Low-Impedance Circuit Element)
A two-terminal, normally open antifuse element. The PLICE is programmed by applying a high voltage across the device and passing current through it. Once programmed, it behaves like a resistor.

PRA, PRB (or MPRA, MPRB)
Device pin. When MODE=1, PRA and PRB function as Actionprobe pins. When MODE=0, PRA and PRB are user-defined I/O pins.

Place TI-ALS program that determines the placement of logic macros in the TPC arrays.

RT/IDLE A Run Test/Idle state in the TAP diagram for idling or running test operations.

Route TI-ALS program that determines the interconnection of logic macros in TPC arrays.

Row A horizontal tile of logic modules; the lowest row in the array is row 0 (zero). Channel n is always below row n.

SDI Device pin. When MODE=0, SDI is a user I/O, and when MODE=1, SDI is used by the program/debug hardware as a serial data input for internal program/test registers.

SDO Device pin. Same as above except SDO in place of SDI and serial data output. TPC12 Series only.

Security Fuse Special antifuses for providing user security for TPC devices after programming.

Segment A piece of wire used for routing. It is connected by one or more antifuses to other segments in either the same or perpendicular direction. Horizontal segments run horizontally in the channels. Vertical segments run vertically through the modules and are often dedicated (connected to a module input or output).

SELDRS A Select Data Register Scan state in the TAP diagram allowing entry into either a data scanning sequence or the SELIRS state.

SELIRS A Select Instruction Register Scan state in the TAP diagram allowing entry into either an instruction scanning sequence or the TLRST state.

SHIFTDR A Shift Data Register state in the TAP diagram allowing data to be shifted through the selected data register from the TDI to TDO pins.

SHIFTIR A Shift Instruction Register state in the TAP diagram allowing data to be shifted through the instruction register from the TDI to TDO pins.

Short Path A chain of a small number of macros (usually less than five) connected by up to three nets. The chain of nets need NOT be a signal path, that is, the path can go from one input pin to another, or the nets can carry signals in opposite directions.

Soft Macro Predefined blocks consisting of multiple hard and/or unit and/or other soft macros. Placement and routing is not predetermined for soft macros.

Stretched Short Path
A short path where the first and the last macros in the chain are fixed macros and are located so far apart that at least one of the nets in this path must use a long horizontal and/or long vertical routing track.

TAP A 16-state finite state machine, referred to as a Test Access Port, that regulates the operation of the 1149.1 boundary scan test architecture.

TMS	Test Mode Select control input to the TAP of an 1149.1 compliant IC.
TCK	Test Clock input to the TAP of an 1149.1 compliant IC.
TDI	Serial Test Data Input to an 1149.1 compliant IC.
TDO	Serial Test Data Output from an 1149.1 compliant IC.
TRSTZ	An optional active low Test Reset input to the TAP of an 1149.1 compliant IC.
TINT	A proposed optional Test Interrupt output from an 1149.1 compliant IC.
TLRST	A Test Logic Reset state in the TAP diagram where the test logic in the IC is placed in an initialized state.
TI-ALS	Texas Instruments Action Logic System.
TPC10	FPGA family consisting of the TPC1010 and TPC1020 devices.
TPC12	FPGA family consisting of the TPC1225, TPC1240, and TPC1280 devices.
TPC14	FPGA family in preliminary form designed as a third generation TI product.
Timer	TI-ALS program for performing static timing analysis on TPC devices.
Track	A set of segments connected end-to-end, running across the array in the vertical or horizontal direction. The segments may or may not be connected by antifuses.
Unit Macro	A cell from the TI macro library implemented in one logic module. A unit-level macro can be implemented in any of several ways by the configuration software.
UPDATEDR	An Update Data Register state in the TAP allowing data shifted into a selected data register to be output in parallel.
UPDATEIR	An Update Instruction Register state in the TAP allowing data shifted into the instruction register to be output in parallel.
User macro	A customer design block consisting of multiple and/or unit hard macros and/or soft macros or other user macros. Placement and routing is not predetermined. *User* is the default if no LEVEL property is attached.
V_{CC}	Device pin. All pins labeled V_{CC} must be connected to the +5 V power supply.
V_{KS}	Device pin. During programming, a *keeper supply* voltage is applied to VKS. During normal operation, this pin must be tied to circuit ground.
V_{PP}	Device pin. During programming, high voltage is applied to V_{PP}. During normal operation, this pin must be tied to V_{CC}.

V_{SV} Device pin. During programming, a *super* voltage is applied to V_{SV}. During normal operation, this pin must be tied to V_{CC}.

Warning A nonfatal message that alerts you to a situation that may cause problems or difficulties if not corrected.

Part 2 – Operating Conditions And Characteristics

> **Note:**
>
> *Current flowing out of* a terminal is given as a negative value.

C_i ***Input capacitance***
The internal capacitance at an input of the device.

C_o ***Output capacitance***
The internal capacitance at an output of the device.

f_{clock} ***Maximum clock frequency***
The highest rate at which the clock input of a bistable circuit can be driven through its required sequence while maintaining stable transitions of logic level at the output with input conditions established that should cause changes of output logic level in accordance with the specification.

I_{CC} ***Supply current***
The *current flowing into* the V_{CC} supply terminal of an integrated circuit.

I_{CCH} ***Supply current, outputs high***
The *current flowing into* the V_{CC} supply terminal of an integrated circuit when all (or a specified number) of the outputs are at the high level.

I_{CCL} ***Supply current, outputs low***
The *current flowing into* the V_{CC} supply terminal of an integrated circuit when all (or a specified number) of the outputs are at the low level.

I_{IH} ***High-level input current***
The *current flowing into* an input when a high-level voltage is applied to that input.

I_{IL} ***Low-level input current***
The *current flowing into* an input when a low-level voltage is applied to that input.

I_{OH}

High-level output current
The *current flowing into* an output with input conditions applied that, according to the product specification, will establish a high level at the output.

I_{OL}

Low-level output current
The *current flowing into* an output with input conditions applied that, according to the product specification, will establish a low level at the output.

I_{OS} (I_O)

Short-circuit output current
The *current flowing into* an output when that output is short-circuited to ground (or other specified potential) with input conditions applied to establish the output logic level farthest from ground potential (or other specified potential).

I_{OZ}

The *current flowing into* an output having 3-state capability with input conditions established that, according to the production specification, will establish the high-impedance state at the output.

Note:

A minimum is specified that is the least-positive value of high-level input voltage for which operation of the logic element within specification limits is guaranteed.

I_{OZH}

Off-state (high-impedance-state) output current (of a 3-state output) with high-level voltage applied
The *current flowing into* an output having 3-state capability with input conditions established that, according to the product specification, will establish the high-impedance state at the output and with a high-level voltage applied to the output.

Note:

This parameter is measured with other input conditions established that would cause the output to be at a low level if it were enabled.

I_{OZL}

***Off-state (high-impedance-state)* output current (of a 3-state output) with low-level voltage applied**
The *current flowing into* an output having 3-state capability with input conditions established that, according to the product specification, will establish the high-impedance state at the output and with a low-level voltage applied to the output.

Note:

This parameter is measured with other input conditions established that would cause the output to be at a high level if it were enabled.

V_{IH}

High-level input voltage
An input voltage within the more positive (less negative) of the two ranges of values used to represent the binary variables.

Note:

A minimum is specified that is the least positive value of high-level input voltage for which operation of the logic element within specification limits is guaranteed.

V_{IK}

Input clamp voltage
An input voltage in a region of relatively low differential resistance that serves to limit the input voltage swing.

V_{IL}

Low-level input voltage
An input voltage within the less positive (more negative) of the two ranges of values used to represent the binary variables.

Note:

A minimum is specified that is the most positive value of low-level input voltage for which operation of the logic element within specification limits is guaranteed.

V_{OH}

High-level output voltage
The voltage at an output terminal with input conditions applied that, according to product specification, will establish a high level at the output.

V_{OL}

Low-level output voltage
The voltage at an output terminal with input conditions applied that, according to product specification, will establish a low level at the output.

t_a

Access time
The time interval between the application of a specific input pulse and the availability of valid signals at an output.

t_{dis}

Disable time (of a 3-state output)
The time interval between the specified reference points on the input and output voltage waveforms, with the 3-state output changing from either of the defined active levels (high or low) to a high-impedance (off) state.
($t_{dis} = t_{PHZ}$ or t_{PLZ}).

t_{en}

Enable time (of a 3-state output)
The time interval between the specified reference points on the input and output voltage waveforms, with the 3-state output changing from a high-impedance (off) state to either of the defined active levels (high or low). (t_{en} = t_{PZH} or t_{PZL}).

t_h

Hold time
The time interval during which a signal is retained at a specified input terminal after an active transition occurs at another specified input terminal.

Note:

1. The hold time is the actual time interval between two signal events and is determined by the system in which the digital circuit operates. A minimum value is specified that is the shortest interval for which correct operation of the digital circuit is guaranteed.
2. The hold time may have a negative value in which case the minimum limit defines the longest interval (between the release of the signal and the active transition) for which correct operation of the digital circuit is guaranteed.

t_{pd}

Propagation delay time
The time between the specified reference points on the input and output voltage waveforms with the output changing from one defined level (high or low) to the other defined level. (t_{pd} = t_{PHL} or t_{PLH}).

t_{PHL}

Propagation delay time, high-to-low level output
The time between the specified reference points on the input and output voltage waveforms with the output changing from the defined high level to the defined low level.

t_{PHZ}

Disable time (of a 3-state output) from high level
The time interval between the specified reference points on the input and the output voltage waveforms with the 3-state output changing from the defined high level to a high-impedance (off) state.

t_{PLH}

Propagation delay time, low-to-high level output
The time between the specified reference points on the input and output voltage waveforms with the output changing from the defined low level to the defined high level.

t_{PLZ}

Disable time (of a 3-state output) from low level
The time interval between the specified reference points on the input and the output voltage waveforms with the 3-state output changing from the defined low level to a high-impedance (off) state.

t_{PZH}

Enable time (of a 3-state output) to high level
The time interval between the specified reference points on the input and output voltage waveforms with the 3-state output changing from a high-impedance (off) state to the defined high level.

t_{PZL}

Enable time (of a 3-state output) to low level
The time interval between the specified reference points on the input and output voltage waveforms with the 3-state output changing from a high-impedance (off) state to the defined low level.

$t_{sk(o)}$

Output Skew
The time interval between any two propagation delay times when a single switching input or multiple inputs switching simultaneously causes multiple outputs to switch, as observed across all switching outputs.

t_{su}

Setup time
The time interval between the application of a signal at a specified input terminal and a subsequent active transition at another specified input terminal.

Note:

1. The setup time is the actual time interval between two signal events and is determined by the system in which the digital circuit operates. A minimum value is specified that is the shortest interval for which correct operation of the digital circuit is guaranteed.
2. The setup time may have a negative value in which case the minimum limit defines the longest interval (between the active transition and the application of the other signal) for which correct operation of the digital circuit is guaranteed.

t_w

Pulse duration (width)
The time interval between specified reference points on the leading and trailing edges of the pulse waveform.

Notes

Notes

TI Worldwide Sales Offices

ALABAMA: Huntsville: 4960 Corporate Drive, Suite 150, Huntsville, AL 35805, (205) 837-7530.
ARIZONA: Phoenix: 8825 N. 23rd Avenue, Suite 100, Phoenix, AZ 85021, (602) 995-1007.
CALIFORNIA: Irvine: 1920 Main Street, Suite 900, Irvine, CA 92714, (714) 660-1200; **San Diego:** 5625 Ruffin Road, Suite 100, San Diego, CA 92123, (619) 278-9600; **Santa Clara:** 5353 Betsy Ross Drive, Santa Clara, CA 95054, (408) 980-9000; **Woodland Hills:** 21550 Oxnard Street, Suite 700, Woodland Hills, CA 91367, (818) 704-8100.
COLORADO: Aurora: 1400 S. Potomac Street, Suite 101, Aurora, CO 80012, (303) 368-8000.
CONNECTICUT: Wallingford: 9 Barnes Industrial Park So., Wallingford, CT 06492, (203) 269-0074.
FLORIDA: Altamonte Springs: 370 S. North Lake Boulevard, Suite 1008, Altamonte Springs, FL 32701, (407) 260-2116; **Fort Lauderdale:** 2950 N.W. 62nd Street, Suite 100, Fort Lauderdale, FL 33309, (305) 973-8502; **Tampa:** 4803 George Road, Suite 390, Tampa, FL 33634-6234, (813) 885-7588.
GEORGIA: Norcross: 5515 Spalding Drive, Norcross, GA 30092-2560, (404) 662-7900.
ILLINOIS: Arlington Heights: 515 West Algonquin, Arlington Heights, IL 60005, (708) 640-2925.
INDIANA: Carmel: 550 Congressional Drive, Suite 100, Carmel, IN 46032, (317) 573-6400; **Fort Wayne:** 103 Airport North Office Park, Fort Wayne, IN 46825, (219) 489-4697.
KANSAS: Overland Park: 7300 College Boulevard, Lighton Plaza, Suite 150, Overland Park, KS 66210, (913) 451-4511.
MARYLAND: Columbia: 8815 Centre Park Drive, Suite 100, Columbia, MD 21045, (410) 964-2003.
MASSACHUSETTS: Waltham: Bay Colony Corporate Center, 950 Winter Street, Suite 2800, Waltham, MA 02154, (617) 895-9100.
MICHIGAN: Farmington Hills: 33737 W. 12 Mile Road, Farmington Hills, MI 48018, (313) 553-1581.
MINNESOTA: Eden Prairie: 11000 W. 78th Street, Suite 100, Eden Prairie, MN 55344, (612) 828-9300.
MISSOURI: St. Louis: 12412 Powerscourt Drive, Suite 125, St. Louis, MO 63131, (314) 821-8400.
NEW JERSEY: Iselin: Metropolitan Corporate Plaza, 485 Bldg E. U.S. 1 South, Iselin, NJ 08830, (908) 750-1050.
NEW MEXICO: Albuquerque: 2709 J. Pan American Freeway, N.E., Albuquerque, NM 87107, (505) 345-2555.
NEW YORK: East Syracuse: 6365 Collamer Drive, East Syracuse, NY 13057, (315) 463-9291; **Fishkill:** 300 Westage Business Center, Suite 140, Fishkill, NY 12524, (914) 897-2900; **Melville:** 48 South Service Road, Suite 100, Melville, NY 11747, (516) 454-6601; **Pittsford:** 2851 Clover Street, Pittsford, NY 14534, (716) 385-6770.
NORTH CAROLINA: Charlotte: 8 Woodlawn Green, Suite 100, Charlotte, NC 28217, (704) 527-0930; **Raleigh:** 2809 Highwoods Boulevard, Suite 100, Raleigh, NC 27625, (919) 876-2725.
OHIO: Beachwood: 23775 Commerce Park Road, Beachwood, OH 44122-5875, (216) 765-7258; **Beavercreek:** 4200 Colonel Glenn Highway, Suite 600, Beavercreek, OH 45431, (513) 427-6200.
OREGON: Beaverton: 6700 S.W. 105th Street, Suite 110, Beaverton, OR 97005, (503) 643-6758.
PENNSYLVANIA: Blue Bell: 670 Sentry Parkway, Suite 200, Blue Bell, PA 19422, (215) 825-9500.
PUERTO RICO: Hato Rey: 615 Mercantile Plaza Building, Suite 505, Hato Rey, PR 00919, (809) 753-8700.
TEXAS: Austin: 12501 Research Boulevard, Austin, TX 78759, (512) 250-6769; **Dallas:** 7839 Churchill Way, Dallas, TX 75251, (214) 917-1264; **Houston:** 9301 Southwest Freeway, Suite 360, Houston, TX 77074, (713) 778-6592; **Midland:** FM 1788 & I-20, Midland, TX 79711-0448, (915) 561-7137.
UTAH: Salt Lake City: 2180 South 1300 East, Suite 335, Salt Lake, UT 54106, (801) 466-8972.
WISCONSIN: Waukesha: 20825 Swenson Drive, Suite 900, Waukesha WI 53186, (414) 798-1001.
CANADA: Nepean: 301 Moodie Drive, Suite 102, Mallorn Center, Nepean, Ontario, Canada K2H 9C4, (613) 726-1970; **Richmond Hill:** 280 Centre Street East, Richmond Hill, Ontario, Canada L4C 1B1, (416) 884-9181; **St. Laurent:** 9460 Trans Canada Highway, **St. Laurent**, Quebec, Canada H4S 1R7, (514) 335-8392.

AUSTRALIA (& NEW ZEALAND): Texas Instruments Australia Ltd., 6-10 Talavera Road, North Ryde (Sydney), New South Wales, Australia 2113, 2-878-9000; 14th Floor, 380 Street, Kilda Road, Melbourne, Victoria, Australia 3004, 3-696-1211; 171 Philip Highway, Elizabeth, South Australia 5112, 8 255-2066.
BELGIUM: Texas Instruments Belgium S.A./N.V., Avenue Jules Bordetlaan 11, 1140 Brussels, Belgium, (02) 242 30 80.
BRAZIL: Texas Instruments Electronicos do Brasil Ltda., Av. Eng. Luiz Carlos Berrini, 1461-11o andar Pinheiros, 04571 Sao Paulo SP, Brazil, 11-535-5133.
DENMARK: Texas Instruments A/S, Borupvang 2D, 2750 Ballerup, Denmark, (44) 68 74 00.
FINLAND: Texas Instruments OY, Ahertajantie 3, P.O. Box 86, 02321 Espoo, Finland, (0) 802 6517.
FRANCE: Texas Instruments France, 8-10 Avenue Morane-Saulnier, B.P. 67, 78141 Velizy-Villacoublay Cedex, France, (1) 30 70 1003.
GERMANY: Texas Instruments Deutschland GmbH., Haggertystrasse 1, 8050 Freising, (08161) 80-0; Kurfurstendamm 195-196, 1000 Berlin 15, (030) 8 82 73 65; Dusseldorfer Strasse 40, 6236 Eschborn 1, (06196) 80 70; Kirchhorster Strasse 2, 3000 Hannover 51, (0511) 64 68-0; Maybachstrasse II, 7302 Ostfildem 2 (Nellingen), (0711) 3403257; Gildehofcenter, Hollestrasse 3, 4300 Essen 1, (0201) 24 25-0.
HOLLAND: Texas Instruments Holland B.V., Hogehilweg 19, Postbus 12995, 1100 AZ Amsterdam-Zuidoost, Holland, (020) 5602911.
HONG KONG: Texas Instruments Hong Kong Ltd., 8th Floor, World Shipping Center, 7 Canton Road, Kowloon, Hong Kong, 737-0338.
HUNGARY: Texas Instruments Representation, Budaorsi ut 42, 1112 Budapest, Hungary, (1) 1 66 66 17.
IRELAND: Texas Instruments Ireland Ltd., 7/8 Harcourt Street, Dublin 2, Ireland, (01) 755233.
ITALY: Texas Instruments Italia S.p.A., Centro Direzionale Colleoni, Palazzo Perseo-Via Paracelso, 12, 20041, Agrate Brianza (Mi), Italy, (039) 63221; Via Castello della Magliana, 38, 00148 Roma, Italy, (06) 6572651; Via Amendola, 17, 40100 Bologna, Italy, (051) 554004.
JAPAN: Texas Instruments Japan Ltd., Aoyama Fuji Building 3-6-12 Kita-Aoyama Minato-ku, Tokyo, Japan 107, 03-498-2111; MS Shibaura Building 9F, 4-13-23 Shibaura, Minato-ku, Tokyo, Japan 108, 03-769-8700; Nissho-Iwai Building 5F, 2-5-8 Imabashi, Chuou-ku, Osaka, Japan 541, 06-204-1881; Dai-ni Toyota Building Nishi-kan 7F, 4-10-27 Meieki, Nakamura-ku, Nagoya, Japan 450, 052-583-8691; Kanazawa Oyama-cho Daiichi Seimei Building 6F, 3-10 Oyama-cho, Kanazawa-shi, Ishikawa, Japan 920, 0762-23-5471; Matsumoto Showa Building 6F, 1-2-11 Fukashi, Matsumoto-shi, Nagano, Japan 390, 0263-33-1060; Daiichi Olympic Tachikawa Building 6F, 1-25-12, Akebono-cho, Tachikawa-shi, Tokyo, Japan 190, 0425-27-6760; Yokohama Business Park East Tower 10F, 134 Goudo-cho, Hodogaya-ku, Yokohama-shi, Kanagawa, Japan 240, 045-338-1220; Nihon Seimei Kyoto Yasaka Building 5F, 843-2, Higashi-Shiokohji-cho, Higashi-iru, Nishinotoh-in, Shiokohji-dori, Shimogyo-ku, Kyoto, Japan 600, 075-341-7713; Sumitomo Seimei Kumagaya Building 8F, 2-44 Yayoi, Kumagaya-shi, Saitama, Japan 360, 0485-22-2440; 2597-1, Aza Harudai, Oaza Yasaka, Kitsuki-shi, Oita, Japan 873, 09786-3-3211.
KOREA: Texas Instruments Korea Ltd., 28th Floor, Trade Tower, 159-1, Samsung-Dong, Kangnam-ku Seoul, Korea, 2-551-2800.
MALAYSIA: Texas Instruments Malaysia, Sdn. Bhd., Asia Pacific, Lot 36.1 #Box 93, Menara Maybank, 100 Jalan Tun Perak, 50050 Kuala Lumpur, Malaysia, 2306001.
MEXICO: Texas Instruments de Mexico S.A. de C.V., Alfonso Reyes 115, Col. Hipodromo Condesa, Mexico, D.F., 06170, 5-515-6081.
NORWAY: Texas Instruments Norge A/S, PB 106, Refstad (Sinsenveien 53), 0513 Oslo 5, Norway, (02) 155 090.
PEOPLE'S REPUBLIC OF CHINA: Texas Instruments China Inc., Beijing Representative Office, 7-05 CITIC Building, 19 Jianguomenwai Dajie, Beijing, China, 500-2255, Ext. 3750.
PHILIPPINES: Texas Instruments Asia Ltd., Philippines Branch, 14th Floor, Ba-Lepanto Building, Paseo de Roxas, Makati, Metro Manila, Philippines, 2-8176031.
PORTUGAL: Texas Instruments Equipamento Electronico (Portugal) LDA., Ing. Frederico Ulricho, 2650 Moreira Da Maia, 4470 Maia, Portugal (2) 948 1003.
SINGAPORE (& INDIA, INDONESIA, THAILAND): Texas Instruments Singapore (PTE) Ltd., Asia Pacific, 101 Thomson Road, #23-01, United Square, Singapore 1130, 3508100.

SPAIN: Texas Instruments Espana S.A., c/Gobelas 43, Urbanizasion La Florida, 28023 Madrid, Spain, (1) 372 8051; c/Diputacion, 279-3-5, 08007 Barcelona, Spain, (3) 317 91 80.
SWEDEN: Texas Instruments International Trade Corporation (Sverigefilialen), Isafjordsgaten, Box 30, 164 93 Kista, Sweden, (08) 752 58 00.
SWITZERLAND: Texas Instruments Switzerland AG, Riedstrasse 6, 8953 Dietikon, Switzerland, (01) 744 2811.
TAIWAN: Texas Instruments Taiwan Limited, Taipei Branch, 10th Floor Bank Tower, 205, Tung Hua N. Road, Taipei, Taiwan, 10592, Republic of China, 2-713 9311.
TURKEY: Texas Instruments, DSEG MidEast Regional Marketing Office, Karum Center, Suite 442, Iran Caddesi 21, 06680 Kavaklidere, Ankara, Turkey, 4-468-0155.
UNITED KINGDOM: Texas Instruments Ltd., Manton Lane, Bedford, England, MK41 7PA, (0234) 270 111.

Worldwide Regional Technology Centers

NORTH AMERICA

ATLANTA: Texas Instruments Incorporated, 5515 Spalding Drive, Norcross, GA 30092, (404) 662-7945.
BOSTON: Texas Instruments Incorporated, 950 Winter Street, Bay Colony Corp. Center, Suite 2800, Waltham, MA 02154, (617) 895-9196.
CHICAGO: Texas Instruments Incorporated, 515 W. Algonquin Road, Arlington Heights, IL 60005, (708) 640-3919.
DALLAS: Texas Instruments Incorporated, 7839 Churchill Way, Park Central South, MS 3984, P.O. Box 650311, Dallas, TX 75251, (214) 917-3881.
INDIANAPOLIS: Texas Instruments Incorporated, 550 Congressional Blvd., Suite 100, Carmel, IN 46032, (317) 573-6400.
IRVINE: Texas Instruments Incorporated, 1920 Main St., Suite 900, Irvine, CA 92714, (714) 660-8140.
MEXICO CITY: Texas Instruments de Mexico, S.A. de C.V., Alfonso Reyes 115, Col. Hipodromo Condesa, Mexico, D.F., 06170, 5-515-6081.
MINNEAPOLIS: Texas Instruments Incorporated, 11000 W. 78th Street, Suite 100, Eden Prairie, MN 55344, (612) 828-9300.
OTTAWA: Texas Instruments Incorporated, 301 Moodie Drive, Suite 102, Nepean, Ontario, Canada K2H 9C4, (613) 726-1970.
SANTA CLARA: Texas Instruments Incorporated, 5353 Betsy Ross Drive, Santa Clara, CA 95054, (408) 748-2222.

ASIA

HONG KONG: Texas Instruments Asia Ltd., 8th Floor World Shipping Center, 7 Canton Road, Kowloon, Hong Kong, 737-0338.
KOREA: Texas Instruments Korea Ltd., Korea Branch, 28th floor, Trade Tower, 159, Samsung-Dong, Kangnam-Ku, Seoul, Korea, 2-551-2800.
SINGAPORE: Texas Instruments Singapore (PTE) Ltd., Asia Pacific Division, 101 Thompson Road, #23-01, United Square, Singapore 1130, 251-9818.
TAIWAN: Texas Instruments Taiwan Ltd., Taipei Branch, 9th Floor, Bank Tower, 205 Tun Hua N. Road, Taipei, Taiwan 10592, Republic of China, 2-713 9311.
TOKYO: Texas Instruments Japan Ltd., MS Shibaura Building 9F, 4-13-23 Shibaura, Minato-ku, Tokyo, Japan 108, 03-3769-8700.

EUROPE

BEDFORD: Texas Instruments Ltd., Manton Lane, Bedford, England MK41 7PA, (0234) 270 111.
FREISING: Texas Instruments Deutschland GmbH., Haggertystrasse 1, 8050 Freising, Federal Republic of Germany, (8161) 80 40 43.
HANNOVER: Texas Instruments Deutschland GmbH., Kirchhorster Strasse 2, 3000 Hannover 51, Federal Republic of Germany, (511) 64 680.
MILAN: Texas Instruments Italia, S.p.A., Centro Direzionale Colleoni, Palazzo Perseo, Via Paracelso 12, 20041, Agrate Brianza (Mi), Italy (039)-63221.
SWEDEN: Texas Instruments International Trade Corporation, Box 30, S-164 93 Kista, Isafjordsgatan 7, Sweden, (08) 752 58 00.
VELIZY (Paris): Texas Instruments France, 8-10 Avenue Morane Saulnier, Borte Postale 67, 78141 Velizy-Villacoublay cedex, France, (1) 30 70 1001.

AUSTRALIA

NEW SOUTH WALES: Texas Instruments Australia Ltd., 6-10 Talavera Road, North Ryde, New South Wales, Australia 2113, 2-878-9000.

SOUTH AMERICA

SAO PAULO: Texas Instruments Electronicos do Brasil Ltda., Av. Eng. Luiz Carlos Berrini, 1461-11o andar, 04571, Sao Paulo, SP, Brazil, 11-535-5133.

C0892

©1992 Texas Instruments Incorporated

TI Worldwide Sales Offices

ALABAMA: Huntsville: 4960 Corporate Drive, Suite 150, Huntsville, AL 35805, (205) 837-7530.

ARIZONA: Phoenix: 8825 N. 23rd Avenue, Suite 100, Phoenix, AZ 85021, (602) 995-1007.

CALIFORNIA: Irvine: 1920 Main Street, Suite 900, Irvine, CA 92714, (714) 660-1200; **San Diego:** 5625 Ruffin Road, Suite 100, San Diego, CA 92123, (619) 278-9600; **Santa Clara:** 5353 Betsy Ross Drive, Santa Clara, CA 95054, (408) 980-9000; **Woodland Hills:** 21550 Oxnard Street, Suite 700, Woodland Hills, CA 91367, (818) 704-8100.

COLORADO: Aurora: 1400 S. Potomac Street, Suite 101, Aurora, CO 80012, (303) 368-8000.

CONNECTICUT: Wallingford: 9 Barnes Industrial Park South, Wallingford, CT 06492, (203) 269-0074.

FLORIDA: Altamonte Springs: 370 S. North Lake Boulevard, Suite 1008, Altamonte Springs, FL 32701, (407) 260-2116; **Fort Lauderdale:** 2950 N.W. 62nd Street, Suite 100, Fort Lauderdale, FL 33309, (305) 973-8502; **Tampa:** 4803 George Road, Suite 390, Tampa, FL 33634-6234, (813) 885-7588.

GEORGIA: Norcross: 5515 Spalding Drive, Norcross, GA 30092-2560, (404) 662-7967.

ILLINOIS: Arlington Heights: 515 West Algonquin, Arlington Heights, IL 60005, (708) 640-2925.

INDIANA: Carmel: 550 Congressional Drive, Suite 100, Carmel, IN 46032, (317) 573-6400; **Fort Wayne:** 103 Airport North Office Park, Fort Wayne, IN 46825, (219) 489-4697.

KANSAS: Overland Park: 7300 College Boulevard, Lighton Plaza, Suite 150, Overland Park, KS 66210, (913) 451-4511.

MARYLAND: Columbia: 8815 Centre Park Drive, Suite 100, Columbia, MD 21045, (410) 964-2003.

MASSACHUSETTS: Waltham: Bay Colony Corporate Center, 950 Winter Street, Suite 2800, Waltham, MA 02154, (617) 895-9100.

MICHIGAN: Farmington Hills: 33737 W. 12 Mile Road, Farmington Hills, MI 48018, (313) 553-1581;

MINNESOTA: Eden Prairie: 11000 W. 78th Street, Suite 100, Eden Prairie, MN 55344, (612) 828-9300.

MISSOURI: St. Louis: 12412 Powerscourt Drive, Suite 125, St. Louis, MO 63131, (314) 821-8400.

NEW JERSEY: Iselin: Metropolitan Corporate Plaza, 485 Bldg. E. U.S. 1 South, Iselin, NJ 08830, (908) 750-1050.

NEW MEXICO: Albuquerque: 2709 J. Pan American Freeway, N.E., Albuquerque, NM 87101, (505) 345-2555.

NEW YORK: East Syracuse: 6365 Collamer Drive, East Syracuse, NY 13057, (315) 463-9291; **Fishkill:** 300 Westage Business Center, Suite 140, Fishkill, NY 12524, (914) 897-2900; **Melville:** 48 South Service Road, Suite 100, Melville, NY 11747, (516) 454-6601; **Pittsford:** 2851 Clover Street, Pittsford, NY 14534, (716) 385-6770.

NORTH CAROLINA: Charlotte: 8 Woodlawn Green, Charlotte, NC 28217, (704) 527-0930; **Raleigh:** 2809 Highwoods Boulevard, Suite 100, Raleigh, NC 27625, (919) 876-2725.

OHIO: Beachwood: 23775 Commerce Park Road, Beachwood, OH 44122-5875, (216) 765-7528; **Beavercreek:** 4200 Colonel Glenn Highway, Suite 600, Beavercreek, OH 45431, (513) 427-6200.

OREGON: Beaverton: 6700 S.W. 105th Street, Suite 110, Beaverton, OR 97005, (503) 643-6758.

PENNSYLVANIA: Blue Bell: 670 Sentry Parkway, Suite 200, Blue Bell, PA 19422, (215) 825-9500.

PUERTO RICO: Hato Rey: 615 Mercantil Plaza Building, Suite 505, Hato Rey, PR 00919, (809) 753-8700.

TEXAS: Austin: 12501 Research Boulevard, Austin, TX 78759, (512) 250-6769; **Dallas:** 7839 Churchill Way, Dallas, TX 75251, (214) 917-1264; **Houston:** 9301 Southwest Freeway, Commerce Park, Suite 360, Houston, TX 77074, (713) 778-6592; **Midland:** FM1788 & I-20, Midland, TX 79711-0448, (915) 561-7137.

UTAH: Salt Lake City: 2180 South 1300 East, Suite 335, Salt Lake City, UT 54106, (801) 466-8972.

WISCONSIN: Waukesha: 20825 Swenson Drive, Suite 900, Waukesha WI 53186, (414) 798-1001.

CANADA: Nepean: 301 Moodie Drive, Suite 102, Mallom Center, Nepean, Ontario, Canada K2H 9C4, (613) 726-1970; **Richmond Hill:** 280 Centre Street East, Richmond Hill, Ontario, Canada L4C 1B1, (416) 884-9181; **St. Laurent:** 9460 Trans Canada Highway, St. Laurent, Quebec, Canada H4S 1R7, (514) 335-8392.

AUSTRALIA (& NEW ZEALAND): Texas Instruments Australia Ltd., 6-10 Talavera Road, North Ryde (Sydney), New South Wales, Australia 2113, 2-878-9000; 14th Floor, 380 Street, Kilda Road, Melbourne, Victoria, Australia 3004, 3-696-1211; 171 Philip Highway, Elizabeth, South Australia 5112, 8 255-2066.

BELGIUM: Texas Instruments Belgium S.A./N.V., Avenue Jules Bordetlaan 11, 1140 Brussels, Belgium, (02) 242 30 80.

BRAZIL: Texas Instruments Electronicos do Brasil Ltda., Av. Eng. Luiz Carlos Berrini, 1461-11o, andar, 04571, Sao Paulo, SP, Brazil, 11-535-5133.

DENMARK: Texas Instruments A/S, Borupvang 2D, 2750 Ballerup, Denmark, (44) 68 74 00.

FINLAND: Texas Instruments OY, Ahertajantie 3, P.O. Box 86, 02321 Espoo, Finland, (0) 802 6517.

FRANCE: Texas Instruments France, 8-10 Avenue Morane-Saulnier, B.P. 67, 78141 Velizy-Villacoublay Cedex, France, (1) 30 70 1003.

GERMANY: Texas Instruments Deutschland GmbH., Haggertystrasse 1, 8050 Freising, (08161) 80-0; Kurfurstendamm 195-196, 1000 Berlin 15, (030) 8 82 73 65; Dusseldorfer Strasse 40, 6236 Eschborn 1, (06196) 80 70; Kirchhorster Strasse 2, 3000 Hannover 51, (0511) 64 68-0; Maybachstrasse II, 7302 Ostfildern 2 (Nellingen), (0711) 3403257; Gildehofcenter, Hollestrasse 3, 4300 Essen 1, (0201) 24 25-0.

HOLLAND: Texas Instruments Holland B.V., Hogehilweg 19, Postbus 12995, 1100 AZ Amsterdam-Zuidoost, Holland, (020) 5602911.

HONG KONG: Texas Instruments Hong Kong Ltd., 8th Floor, World Shipping Center, 7 Canton Road, Kowloon, Hong Kong, 737-0338.

HUNGARY: Texas Instruments Representation, Budaorsi ut.42, 1112 Budapest, Hungary, (1) 1.66 66 17.

IRELAND: Texas Instruments Ireland Ltd., 7/8 Harcourt Street, Dublin 2, Ireland, (01) 755233.

ITALY: Texas Instruments Italia S.p.A., Centro Direzionale Colleoni, Palazzo Perseo-Via Paracelso 12, 20041 Agrate Brianza (Mi), Italy, (039) 63221; Via Castello della Magliana, 38, 00148 Roma, Italy (06) 6572651; Via Amendola, 17, 40100 Bologna, Italy (051) 554004.

JAPAN: Texas Instruments Japan Ltd., Aoyama Fuji Building 3-6-12 Kita-Aoyama Minato-ku, Tokyo, Japan 107, 03-498–2111; MS Shibaura Building 9F, 4-13-23 Shibaura, Minato-ku, Tokyo, Japan 108, 03-769-8700; Nissho-iwai Building 5F, 2-5-8 Imabashi, Chuou-ku, Osaka, Japan 541, 06-204-1881; Dai-ni Toyota Building Nishi-kan 7F, 4-10-27 Meieki, Nakamura-ku, Nagoya, Japan 450, 052-583-8691; Kanazawa Oyama-cho Daiichi Seimei Building 6F, 3-10 Oyama-cho, Kanazawa-shi, Ishikawa, Japan 920, 0762-23-5471; Matsumoto Showa Building 6F, 1-2-11 Fukashi, Matsumoto-shi, Nagano, Japan 390, 0263-33-1060; Daiichi Olympic Tachikawa Building 6F, 1-25-12, Akebono-cho, Tachikawa, Tokyo, Japan 190, 0425-27-6760; Yokohama Business Park East Tower 10F, 134 Goudo-cho, Hodogaya-ku, Yokohama-shi, Kanagawa, Japan 240, 045-338-1220; Nihon Seimei Kyoto Yasaka Building 5F, 843-2, Higashi Shiokohji-cho, Higashi-iru, Nishinotoh-in, Shiokohji-dori, Shirmogyo-ku, Kyoto, Japan 600, 075-341-7713; Sumitomo Seimei Kumagaya Building 8F, 2-44 Yayoi, Kumagaya-shi, Saitama, Japan 360, 0485-22-2240; 2597-1, Aza Harudai, Oaza Yasaka, Kitsuki-shi, Oita, Japan 873, 09786-3-3211.

KOREA: Texas Instruments Korea Ltd., 28th Floor, Trade Tower, 159, Samsung-Dong, Kangnam-ku Seoul, Korea, 2-551-2800.

MALAYSIA: Texas Instruments, Malaysia, Sdn. Bhd., Asia Pacific, Lot 36.1 #Box 93, Menara Maybank, 100 Jalan Tun Perak, 50050 Kuala Lumpur, Malaysia, 2306001.

MEXICO: Texas Instruments de Mexico S.A. de C.V., Alfonso Reyes 115, Col. Hipodromo Condesa, MExico, D.F., 06170, 5-515-6081.

NORWAY: Texas Instruments Norge A/S, P.B. 106, Refstad (Sinsenveien 53), 0513 Oslo 5, Norway, (02) 155 090.

PEOPLE'S REPUBLIC OF CHINA: Texas Instruments China Inc., Beijing Representative Office, 7-05 CITIC Building, 19 Jianguomenwai Dajie, Beijing, China, 500-2255, Ext. 3750.

PHILIPPINES: Texas Instruments Asia Ltd., Philippines Branch, 14th Floor, Ba-Lepanto Building, Paseo de Roxas, Makati, Metro Manila, Philippines, 2-8176031.

PORTUGAL: Texas Instruments Equipamento Electronico (Portugal) LDA., Ing. Frederico Ulricho, 2650 Moreira Da Maia, 4470 Maia, Portugal (2) 948 1003.

SINGAPORE (& INDIA, INDONESIA, THAILAND): Texas Instruments Singapore (PTE) Ltd., Asia Pacific, 101 Thomson Road, #23-01, United Square, Singapore 1130, 3508100.

SPAIN: Texas Instruments Espana S.A., c/Gobelas 43, Urbanizasion La Florida, 28023, Madrid, Spain, (1) 372 8051; c/Diputacion, 279-3-5, 08007 Barcelona, Spain, (3) 317 91 80.

SWEDEN: Texas Instruments International Trade Corporation (Sverigefilialen), Isafjordsgatan Box 30, 164 93 Kista, Sweden, (08) 752 58 00.

SWITZERLAND: Texas Instruments Switzerland AG, Riedstrasse 6, 8953 Dietikon, Switzerland, (01) 744 2811.

TAIWAN: Texas Instruments Taiwan Limited, Taipei Branch, 10th Floor, Bank Tower, 205 Tung Hua N. Road, Taipei, Taiwan, 10592, Republic of China, 2-713 9311.

TURKEY: Texas Instruments, DSEG MidEast Regional Marketing Office, Karum Center, Suite 442, Iran Caddesi 21, 06680 Kavaklidere, Ankara, Turkey, 4-468-0155.

UNITED KINGDOM: Texas Instruments Ltd., Manton Lane, Bedford, England, MK41 7PA, (234) 270 111.

A089:

SRFA001