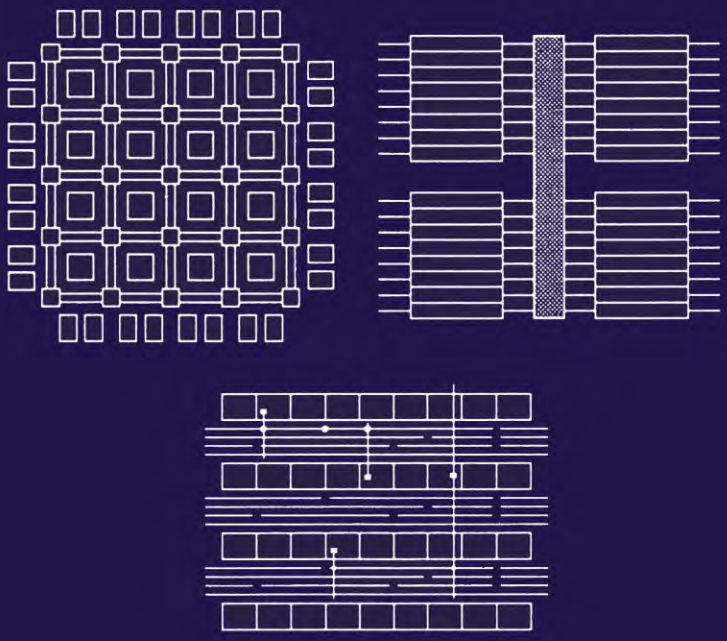
Appendix C.5

Intel Ex. 1009 – Part 4
Intel v. Iida

FIELD-PROGRAMMABLE GATE ARRAY TECHNOLOGY



edited by

Stephen M. Trimberger

FIELD-PROGRAMMABLE GATE ARRAY TECHNOLOGY

FIELD-PROGRAMMABLE GATE ARRAY TECHNOLOGY

edited by

Stephen M. Trimberger
Xilinx

with contributions by

Stephen M. Trimberger
Xilinx

Dennis McCarty
Telle Whitney
Actel

and
The Technical Staff of Altera Corporation
edited by
Robert Hartmann



SPRINGER SCIENCE+BUSINESS MEDIA, LLC

Library of Congress Cataloging-in-Publication Data

Field -programmable gate array technology / edited by Stephen M. Trimberger.
p. cm.
Includes bibliographical references and index.
ISBN 978-1-4613-6183-1 ISBN 978-1-4615-2742-8 (eBook)
DOI 10.1007/978-1-4615-2742-8
1. Gate array circuits. 2. Programmable logic devices.
3. Programmable array logic. I. Trimberger, Stephen, 1955 -
TK7895.G36F54 1994
621.39'5--dc20 93-39703
CIP

Copyright © 1994 by Springer Science+Business Media New York
Originally published by Kluwer Academic Publishers in 1994
Softcover reprint of the hardcover 1st edition 1994
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, mechanical, photo-copying, recording, or otherwise, without the prior written permission of the publisher, Springer Science+Business Media, LLC.

Printed on acid-free paper.

to ross

who had a vision

Contents

Preface xi

Chapter 1. Introduction 1

1.1. Logic Implementation Options 1

1.2. What is an FPGA? 2

1.3. Advantages of FPGAs 4
Low Tooling Costs 4
Rapid Turnaround 4
Low Risk 5
Effective Design Verification 6
Low Testing Costs 6
Standard-Product Advantages 7
Life Cycle Advantages 8

1.4. Disadvantages of FPGAs 8
Chip Size and Cost 8
Speed of Circuitry 9
Design Methodology 9

1.5. Technology Trends 10
Density 10
Speed 10
Architecture 11

1.6. Designing for FPGAs 11
Design Migration 11

1.7. Outline of Subsequent Chapters 12
1. Introduction 12
2. Programming Technology 13
3. Device Architecture 13
4. Software 13
5. The Future 13
6. Design Applications 13
7. Acknowledgments 13
8. References 13

1.8. References 13

Chapter 2. SRAM Programmable FPGAs 15

2.1. Introduction 15

2.2. Programming Technology 15
SRAM Programming 15
Advantages and Disadvantages of SRAM Programming 17

2.3. Device Architecture 19
Simple SRAM-Programmable FPGA Architecture 19
Design Trade-offs 23
The Xilinx XC2000 Architecture 29
The Xilinx XC3000 Architecture 35
The Xilinx XC4000 Architecture 43
Programming the FPGA 52

2.4. Software 53
Automated Design Implementation 54
Technology-Specific Synthesis 63
Manual Design 63

2.5. The Future 65
Programming Technology 65
Architecture 66
Software 66
Partitioning in Space and Time 67
Design Methodology 67

2.6. Design Applications 68
General Design Issues 68
Counter Examples 70
Efficient Multiplication by a Constant in an Artificial Neural Network 75
Distributed Arithmetic for Signal Processing 77
Applications of Reprogramming 79
A Fast Video Controller 83
A Position Tracker For a Robot Manipulator 84
A Fast DMA Controller 85
Custom Computing Applications 87

2.7. Acknowledgments 90

2.8. References 91

Chapter 3. Antifuse Programmed FPGAs 97

3.1 Introduction 97

3.2 Programming Technology 99

3.3 Device Architecture 103
Principles of Programmable Routing 103
Routing Architecture of the Actel FPGAs 108
Act1 Architecture 110
Act2 Architecture 113
Act3 Architecture 117
Programming and Testing 118
Capacity 124
Performance 127

3.4 Software 128

3.5 The Future 132

3.6 Design Applications 133
Designing with ACT1 and ACT2 FPGAs 133
Designing with ACT FPGAs: A TTL Perspective 137
Migrating PLD Designs to FPGAs 140
Synthesis Design Flow 143
Designing Counters with ACT Devices 144
Designing Adders and Accumulators with the ACT Architecture 153
State Machine Design 160
Using FPGAs for Digital PLLs 164
Customer Design Examples 167

3.7 Acknowledgments 168

3.8 References 168

Chapter 4. Erasable Programmable Logic Devices 171

4.1. Introduction 171

4.2. Programming Technology 173
Logic Structures Using EPROM Transistors 175

4.3. Device Architecture **179**
Basic Concepts 179
Macrocell Architecture 180
Logic Array 180
Programmable Flip-Flops 181
Programmable Clock 182
I/O Control Block 182
Design Security 182
Functional Testing 183
Operating Requirements for EPLDs 183
Architectural Evolution in Array-Based PLDs 184
4.3.1 - The "Classic" Family of PLDs 184
Functional Description of the EP1810 184
4.3.2 - The MAX (Multiple Array matriX) Product Family 187
4.3.3 - MAX 7000 195
4.3.4 - MPLDs: Mask-Programmed Logic Devices 200

4.4. Software **204**

4.5. The Future **218**

4.6. Design Applications **224**
4.6.1 MAX 5000 Timing 224
4.6.2 Using Expanders to Build Registered Logic in MAX EPLDs 228
4.6.3 Simulating Internal Buses in General-Purpose EPLDs 233
4.6.4 Fast Bus Controllers with the EPM5016 238
4.6.5 Micro Channel Bus Master and SDP Logic with the EPM5032 EPLD 240
4.6.6 FIFO Controller Using an EPM7096 243
4.6.7 Integrating an Intelligent I/O Subsystem with a Single EPM5130 EPLD 246
4.6.8 Controlling Complex CCD Imaging Systems with the EPS464 EPLD 247

4.7. References **250**

Index **253**

Preface

A Field Programmable Gate Array (FPGA) is a programmable logic device that implements multi-level logic. FPGAs resemble traditional mask-programmed gate arrays by their modular, extensible structure that includes both logic and interconnect, but differ in that their programming is done by end users at their site. No masking steps are required. In this respect, FPGAs resemble PLDs. FPGAs offer low risk, low incremental cost and fast prototyping advantages.

FPGAs are revolutionizing the way systems designers implement logic. By radically reducing the development costs and the turnaround time for implementing thousands of gates of logic, FPGAs provide a new capability that affects the semiconductor industry and the CAE industry. They may also change the way digital systems will be designed in the future.

The Scope of the Book

The field of FPGAs is varied and dynamic. Many different kinds of FPGAs exist, with different programming technologies, different architectures and different software. This book describes the major FPGA architectures available today, covering the three programming technologies that are in use and the major architectures built on those programming technologies. The goal is to introduce the reader to concepts relevant to the entire field of FPGAs using popular devices as examples, without trying to enumerate every commercially-available product.

This book includes discussions of FPGA integrated circuit manufacturing, circuit design and logic design. It describes the way logic and interconnect are implemented in various kinds of FPGAs. It covers particular problems with design for FPGAs and future possibilities for new architectures and software. This book compares CAD for FPGAs with CAD for traditional gate arrays. It describes algorithms for placement, routing and optimization of FPGAs.

The FPGA device descriptions in this book include specifications of capacity and speed. These numbers are continually being debated by manufacturers. This book does not attempt to enter the debate; there was no attempt to reconcile device specifications from different vendors.

FPGA devices and technology are improving rapidly, so specific numbers for gate counts and device speeds may already be obsolete. However, the general concepts,

such as programming methods, architectural constraints due to programming technologies, device scaling and preferred design methods will remain relevant long after the specific devices in this book have only historical interest.

Intended Audience

This book is intended to describe all aspects of FPGA design and development. For this reason, it covers a significant amount of material. An extremely detailed discussion of all these areas would make this book prohibitively long. Our intent is to make each section clear to readers with general technical expertise in digital design and design tools. Readers with significant experience in one of these areas may find the discussions superficial in that area, but useful in others.

This book assumes the reader has an understanding of the fundamentals of digital electronics design. Experience designing or using ASIC gate arrays and software will make much of this book much easier to read, since many of the comparisons are with respect to gate arrays.

Potential developers of FPGAs will benefit primarily from the FPGA architecture and software discussion. Electronics systems designers and ASIC users will find this book gives them a background on different types of FPGAs and shows applications of their use, which are useful for deciding when an FPGA is appropriate for an application.

This book may be useful in a university setting where it can be used in support of a comparative FPGA architectures course, as background reading for a digital design course with FPGAs as the target implementation, or as supplemental reading for a Computer-Aided Design course for tools targeted to FPGA design automation.

This book is not intended as a product specification for any integrated circuit or software product.

Organization of the Book

Chapter 1 introduces the FPGA in comparison with other logic implementation techniques. It defines the term FPGA in a form that is both general enough to include all types of devices currently being offered, and specific enough to be a guide for evaluating other devices that may appear. The bulk of chapter 1 is a comparison of FPGAs with mask programmed gate arrays, showing the FPGA advantages and disadvantages.

The following three chapters describe three very different FPGA architectures and software. Chapter 2 describes Xilinx SRAM-based FPGAs, chapter 3 describes Actel antifuse-based FPGAs and chapter 4 describes Altera EEPROM-based FPGAs. The three architectures were chosen because they were the most common FPGAs currently in use and because they are very different in their approaches to field programmable logic. They have different programming technologies, different

methods of implementing logic and different interconnection strategies. Each of these chapters includes a discussion of an FPGA family, its architecture, software, and applications.

Each chapter was written by an expert in that particular type of architecture. Each author expresses the concepts in the terminology familiar to developers and users of that architecture. To facilitate comparison of the different FPGAs, the chapters follow a common outline, described in chapter 1. In addition, the reader may use the index as a glossary, as terms with similar meanings are correlated there.

Acknowledgments

I would like to thank all those who contributed directly and indirectly to the success of this book, especially the good folks at Xilinx who allowed me the time for this project, especially Bernie Vonderschmitt, Wes Patterson, Gary Leive and Bill Carter. I wish also to thank the authors of the architecture sections, whose effort and endurance were vital to the completion of this project.

Stephen Trimberger
San Jose, CA

Trademarks

Xilinx, XACT, XC2064, XC3090, XC4005, and XC-DS501 are registered trademarks of Xilinx. All XC-prefix product designations, XACT-Performance, XAPP, X-BLOX, XSI, XChecker, XDM, XEPLD, XFT, XAPP, XSI, BITA, Dual Block, FastCLK, HardWire, LCA, Logic Cell, PLUSASM and UIM are trademarks of Xilinx. The Programmable Logic Company is a service mark of Xilinx.

Act is a trademark and Actel, Action Logic, Activator, Actionprobe and PLICE are registered trademarks of Actel Corporation.

Altera, MAX, and MAX+PLUS are registered trademarks of Altera Corporation. The following are trademarks of Altera Corporation: MAX+PLUS II, FastTrack, FLEX, AHDL, MPLD, MAX 5000. MAX 7000, FLEX 8000, Classic, STG, PLS-FLEX, PLDS-HPS, PLDS-MAX, PLS-WS/SN, PLS-WS/HP, PLS-EDIF. Product design elements and mnemonics are Altera Corporation copyright.

ABEL is a trademark of Data I/O Corporation.

Viewlogic is a registered trademark of Viewlogic Systems, Incorporated.

OrCAD is a trademark of OrCAD Systems Corporation.

IBM and AT are registered trademarks and IBM PC, XT, PS/2 and Micro Channel are trademarks of International Business Machines Corporation.

Windows is a trademark of Microsoft Corporation.

Sun is a trademark of Sun Microsystems, Incorporated.

PAL and PALASM are registered trademarks of Advanced Micro Devices, Incorporated.

Synopsys and Design Compiler are trademarks of Synopsys, Inc.

Radius is a trademark and Pivot is a registered trademarks of Radius, Inc.

Apple and Macintosh are registered trademarks of Apple Computer, Inc.

Quickturn and RPM Logic Emulator are a trademarks of Quickturn Design Systems.

All trademarks are the property of their respective owners.

FIELD-PROGRAMMABLE GATE ARRAY TECHNOLOGY

Chapter 1
Introduction

1.1. Logic Implementation Options

An electronic system designer has several options for implementing digital logic. These options include discrete logic devices, often called Small-Scale Integrated circuits, or SSI; programmable devices such as Programmable Array Logic (PALs or PLDs); masked-programmed Gate Arrays or Cell-Based ASICs; and Field Programmable Gate Arrays (FPGAs).

Small amounts of logic can be implemented easily with discrete devices. Each SSI chip contains a few identical gates of a specific type. Designers choose the logic they want from the selection of available chip types. SSI logic is often referred-to as "7400-series," in reference to the widely-used Texas Instruments logic family.

A simple Programmable Logic Device (PLD) is a general-purpose device capable of implementing the logic of tens or hundreds of SSI packages.Pioneered by MMI, a PLD implements logic as wide fan-in two-level sum-of-products of its inputs. It may have optional flip-flops or other logic on the outputs of the sum-of-products array. The best-known PLD is the "22V10", with 22 inputs and 10 outputs, developed by AMD and copied by numerous others. A PLD is programmed by users at their site using inexpensive programming hardware. Power consumption and delay limit the size of the simple sum of products structure to dozens of product terms. Large designs require a multi-level logic implementation.

To implement designs with thousands or tens of thousands of gates, designers can use a Mask Programmed Gate Array (MPGA), commonly called a *gate array*. An MPGA can implement tens of thousands or even hundreds of thousands of gates of logic on a single IC in multi-level logic with wiring between logic stages. An MPGA consists of a base of pre-designed transistors with customized wiring for each design. The wiring is built during the manufacturing process, so each design requires custom masks for the wiring. The mask-making charges make low-volume MPGAs expensive. Typical turnaround times for MPGAs are four to six weeks.

Field Programmable Gate Arrays offer the benefits of both programmable logic arrays and gate arrays. Like MPGAs, FPGAs implement thousands of gates of logic in a single integrated circuit. Like PLDs, FPGAs are programmable by designers at their site,

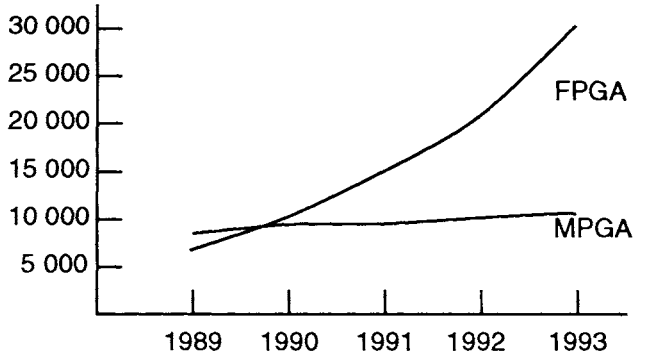


Figure 1.1. Number of Designs Implemented as MPGA Versus FPGA (source: Dataquest, 1991 and Xilinx, 1992)

eliminating the long delays and tooling costs. These advantages have made FPGAs very popular (figure 1.1).

1.2. What is an FPGA?

An FPGA is a general-purpose, multi-level programmable logic device that is customized in the package by the end users. FPGAs are composed of blocks of logic connected with programmable interconnect. The programmable interconnect between blocks allows users to implement multi-level logic, removing many of the size limita-

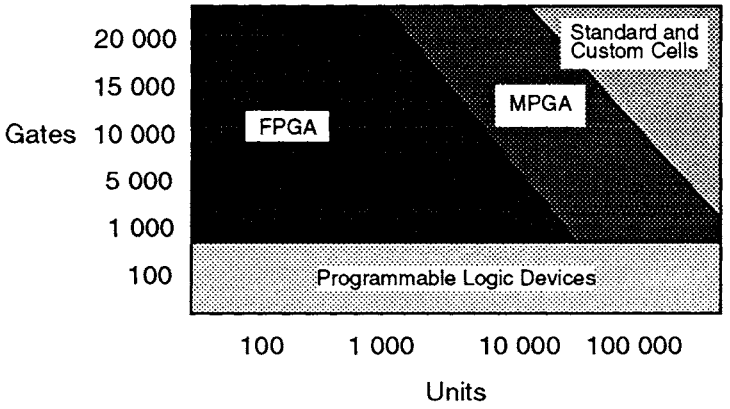


Figure 1.2. Preferred Implementation Options in the Design Space (Source: Xilinx).

tions of the PLD-derived two-level logic structure. This extensible architecture can currently support thousands of gates of logic at system speeds in the tens of megahertz.

The size, structure and number of blocks; and the amount and connectivity of the interconnect vary considerably among FPGA architectures. This difference in architectures is driven by different programming technologies and different target applications of the parts. An architectural organization that works well with a particular programming technology typically does not work with another. The segmentation by programming style and hence architecture is the basis of the taxonomy in figure 1.3. FPGAs fall into four groups: island-style and cellular SRAM-programmed devices; channeled, antifuse-programmed devices; and array-style EPROM or EEPROM-programmed devices.

SRAM-programmed island-style FPGAs include all three Xilinx LCA families, the AT&T Orca and Altera Flex, as well as UTFPGA1 [Chow 1991]. Cellular-style FPGAs include Toshiba, Plessey's ERA, Atmel's (formerly Concurrent Logic) CLi family, the Algotronix CAL, as well as Triptych [Ebeling 1991]. Antifuse-based channelled gate arrays include Actel's ACT-1 and ACT-2, Quicklogic's pASIC and Crosspoint's CP20K Series FPGA. EPROM-programmed array-like devices resemble a collection of PALs with a central interconnection mechanism. Devices of this type are Altera's MAX 5000 and MAX 7000, AMD's Mach and Xilinx's EPLD, among others.

This definition of FPGA is similar to the Complex PLD (CPLD) definition used by the Dataquest market research company. They divide CPLDs into "Programmable Multi-level Devices" (PMDs), which are simple PLD arrays with a programmable

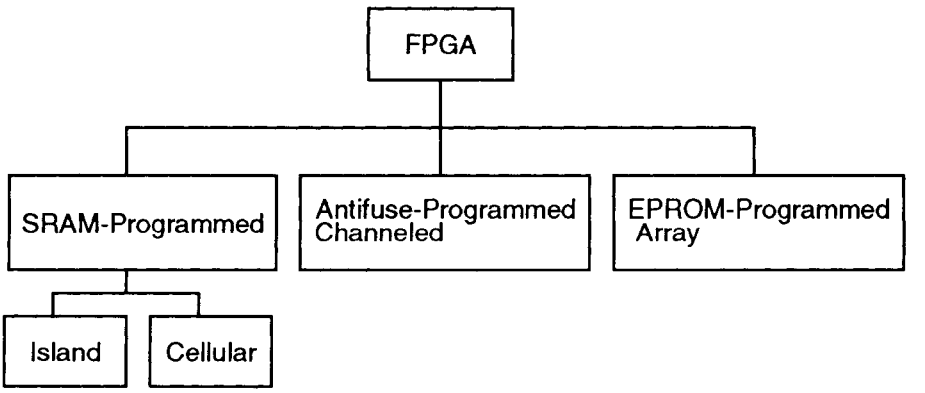


Figure 1.3. Taxonomy of FPGAs.

interconnection structure; and "FPGAs", which consists of all other multi-level field-programmable devices. The Dataquest "FPGA" classification covers those devices in the SRAM-Programmed and Antifuse-Programmed boxes in figure 1.3. In this book, we use the broader definition of FPGAs, which is equivalent to the Dataquest "CPLD" classification.

Currently-available FPGAs implement digital logic, but this is not a fundamental limitation. FPGAs composed of analog blocks with programmable interconnect have been proposed and built [Lee 1991], but they are not commercially available. This book does not describe analog or hybrid FPGAs. The reader is directed to the references for further information on analog FPGAs, also called Field-Programmable Analog Arrays (FPAAs).

1.3. Advantages of FPGAs

Figure 1.4 compares MPGA and FPGA design and manufacturing steps. Design entry and verification are similar for both technologies, but there are significant differences late in the design cycle. Instead of customizing the part by custom manufacturing steps, FPGAs are customized by electrical modification of a packaged part. By eliminating the customization during manufacturing, FPGAs eliminate each design's custom mask-making, test pattern generation, wafer fabrication, packaging and testing. The electrical modification takes milliseconds or minutes, depending on the programming technology and size of the part, compared to weeks for the MPGA steps. FPGA programming is done by simple, inexpensive programming devices.

MPGA	FPGA
System Design	System Design
Logic Design	Logic Design
Place and Route	Place and Route
Timing Simulation	Timing Simulation
Test Pattern Generation	
Mask Making	
Wafer Fabrication	Download / programming
Packaging	
Testing	
System Integration	System Integration

Figure 1.4. Design Steps For MPGA versus FPGA

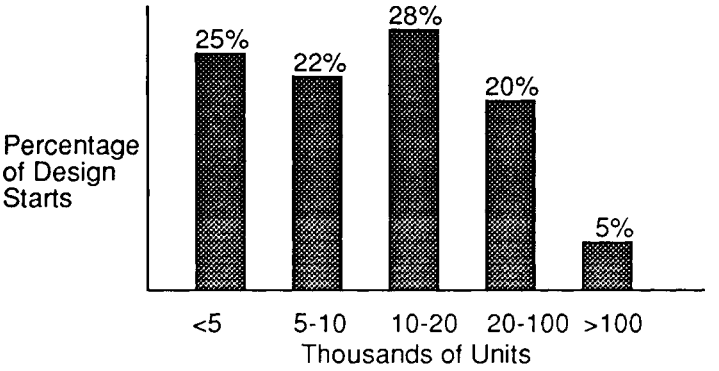


Figure 1.5. MOS Gate Array Design Starts by Unit Volume (source: Dataquest 1991).

Low Tooling Costs

Every design to be implemented in an MPGA requires custom masks to build the custom wiring patterns. Each mask costs several thousand dollars and the cost is amortized over the total number of units manufactured. The more units built, the lower the impact of the masking charges. However, comparatively few designs require more than tens of thousands of units (figure 1.5), so, for most MPGA designs, the masking charges are significant. There is no custom tooling required for an FPGA, so there are no associated tooling costs, making FPGAs cost effective for most logic designs.

Rapid Turnaround

The MPGA manufacturing process takes several weeks from the completion of the design to the delivery of the finished parts. An FPGA can be programmed in minutes by the user of the part. On an FPGA, a modification to correct a design flaw or to address a late specification change can be made quickly and cheaply. Faster design turnaround leads to faster product development and shorter time-to-market for new FPGA products. Reinertsen [1983] determined that in a high-technology environment, a six-month delay in product delivery cut the lifetime profits of a product by thirty-three percent.

Low Risk

The benefits of low initial Non-Recurring Engineering (NRE) charges and rapid turnaround means that a design iteration due to an error incurs neither a large expense nor a long delay. Low costs encourage early system integration and prototyping. The low cost of error also encourages more aggressive logic design, which may yield better

performance and more cost effective designs.

Effective Design Verification

Because of substantial NRE costs and manufacturing delays, MPGA users verify their designs by extensive simulation before manufacture. Simulation has an inherent speed/accuracy trade-off: highly accurate simulators are slow, fast simulators are inaccurate. To verify the functionality of the design in a system, large amounts of time must be simulated. Proper verification requires that the environment of the design be simulated as well. Week-long simulation runs are not uncommon. An MPGA design, verified by simulation, may include errors due to inaccuracies or over-simplifications in the simulation model.

FPGAs avoid these problems. Instead of simulating large amounts of time, FPGA users may choose to use in-circuit verification. Designers can implement the design and use a functioning part as a prototype. The prototype operates at full speed and with excellent timing accuracy. A prototype can be inserted into the system to verify functionality of the system as a whole, eliminating a class of system errors early.

Low Testing Costs

All ICs must be tested to verify proper manufacturing and packaging. This test is different for each design. Designs implemented in an MPGA incur three costs associated with testing: on-chip logic to facilitate testing, generation of the test program and testing the parts when manufacturing is complete. FPGAs address all these costs.

Good test programs are hard to write, and schedule pressures tend to abbreviate test program generation for MPGAs. The poor coverage of MPGA test programs allows some bad chips to pass the testing. These defective parts may not be discovered until they fail in a system where the cost of repair is high.

In contrast, the test program for FPGAs is the same for all designs and tests the FPGA for all users of the part. Because there is only one test program, it is reasonable to invest a considerable amount of effort in it, and it can be continually improved over the lifetime of the FPGA. The resulting test program achieves excellent test coverage, leading to high-quality ICs.

In the case of reprogrammable parts, the manufacturer can reprogram all programmable points during testing to verify that the part will work properly after programming. For one-time-programmable parts, the FPGAs generally include test circuitry to catch most failures during manufacture. Manufacturer-supplied hardware and software verify post-programming functionality. Those parts that fail to pass the post-programming test are rejected on the programming device. The percentage of successfully-programmed devices is termed *programming yield.*

The manufacturer's test program verifies that every FPGA will be functional for all possible designs that may be implemented on it. FPGA users are not required to write

design-specific tests for their designs. Therefore, designers need not build the testability into the design, eliminating "design for testability" and the design effort and overhead associated with it.

Standard-Product Advantages

New, denser integrated circuit technologies drive microelectronic advances. New processing technologies have finer geometries with smaller transistors and wires. The speed and cost of a chip are related to these dimensions, so a smaller chip is both cheaper and faster.

Moving an MPGA design to a new process incurs additional NRE charges for new masks and test program verification, so it is rarely done. Because FPGAs are standard products, only the FPGA manufacturer incurs the cost of moving the chip to a new process technology. FPGAs on the new process are available to all customers without additional NRE cost. From the point of view of a user, the FPGA manufacturer lowers the price and improves the speed of the parts over time. A user of MPGAs does not get these improvements without paying the additional NRE charges.

Normal variations in the integrated circuit manufacturing process leads to a distribution of performance of integrated circuits. In an MPGA, the customer must design to worst-case process characteristics. The chips that meet "typical" specifications rather than "worst-case" specifications are approximately twenty percent faster. FPGA manufacturers can separate the fast parts from the slow ones in a process called *speed binning*. Slower parts sell for lower price. Faster parts allow designers to design to the high-end of the process variation, giving FPGA users a price/performance trade-off that MPGA designers do not have.

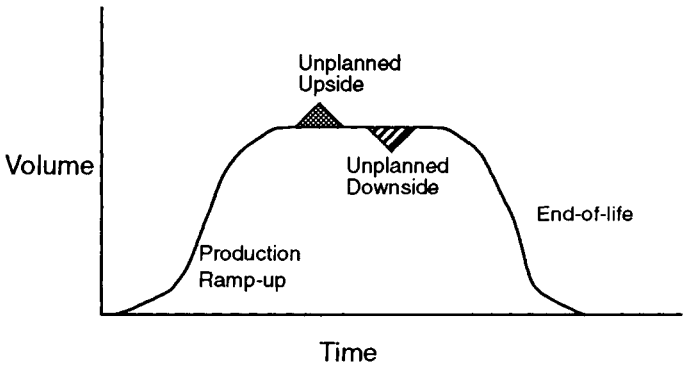


Figure 1.6. Generic Product Life Cycle (source: Xilinx).

Life Cycle Advantages

The life of a product does not end when the design of a chip inside it is finished. Figure 1.6 shows a typical product life cycle plotted as volume versus time. When the design is complete, there is a ramp-up into production. This ramp-up may include a few prototypes or larger pilot manufacturing runs. During production, a design may have periods of increased or decreased sales. At the end of a product lifetime, production tapers off.

MPGA are only cost effective when ordered in volume, and the volume must be decided months in advance of delivery due to long manufacturing cycle times. An MPGA user must maintain a sufficient inventory to handle upturns, and is left with excess inventory should sales fail to meet expectations. At the end of a product lifetime, MPGA users are faced with a last-purchase decision that must be made months in advance of the end-of-product date. If they order too many parts, they are left with unusable parts in inventory, if they order too few, they may not have enough parts to build the last few systems.

The cost-effectiveness of FPGAs in low volume and the flexibility provided by field-programmability provide advantages over all phases of product lifetime. When introducing a product, an FPGA user may order a few parts at a time while testing the design for functionality and the product for market viability. During production, the FPGA user can accommodate rapid changes in sales easily because long lead times are not required. An FPGA user can make product enhancements by shipping an upgraded design on the same FPGA device. This upgrade requires no inventory changes, no new hardware and does not interrupt production.

1.4. Disadvantages of FPGAs

FPGAs have on-chip *programming overhead* circuitry that manages the programming of the part. The area of the programming overhead cannot be used by customers, and lowers the FPGA gate density. The programmable switches and options in an FPGA are larger than the mask programming that can be built in an MPGA. The programmable switches also increase signal delay by adding resistance and capacitance to interconnect paths. As a result, FPGAs are larger and slower than equivalent MPGAs.

Chip Size and Cost

The area penalty for field-programmability is significant. Current FPGAs are about ten times larger for the same gate capacity as the equivalent MPGA, and are correspondingly more expensive on a per-chip basis.

Because of the overhead of field programmability, current-generation FPGAs are limited to tens of thousands of gates of capacity, while the largest MPGAs are hundreds of thousands of gates. For large designs, designers must either split the design into several FPGAs or they must move the design to an MPGA. Multiple-chip partitioning

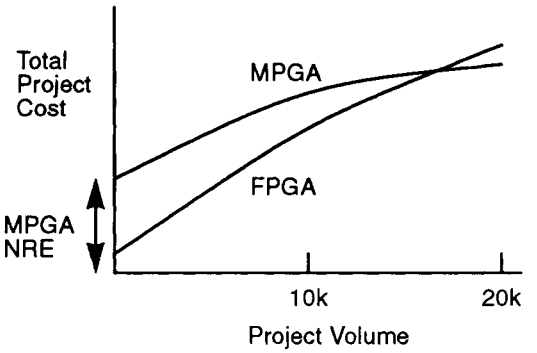


Figure 1.7. Cost Versus Volume for MPGAs vs. FPGAs (source: Xilinx).

for FPGA designs is available, but still relatively immature.

Figure 1.7 shows a cost comparison of MPGAs against FPGAs over a range of volume for the parts. At some point the initial cost savings from the lack of NRE charge with FPGAs is consumed by the increased per-chip costs. That crossover point for a five-thousand-gate part is currently above ten thousand units. Designs with greater volume are more cost effective using an MPGA, despite the greater start-up costs.

Speed of Circuitry

The connection paths in an FPGA are slowed by the programming circuitry. Programmable interconnection points along a wiring path add resistance to the path. All programming points in the interconnect add capacitance to the internal paths. Finally, since more area is required for the same amount of logic, interconnect lines between logic are longer. Longer lines have greater resistance and capacitance, further slowing the resulting circuitry. Current FPGAs are two to three times slower than MPGAs and it is unlikely that they will ever equal MPGA performance on the same manufacturing process.

Design Methodology

FPGAs have been criticized because they are *too easy* to use, thereby encouraging a "try-it-and-see-what-happens" methodology for logic design. If these "sloppy" design practices result in poor-quality designs, the resulting products will be inferior.

1.5. Technology Trends

This section examines the relative performance of FPGAs and MPGAs in the context of advancing process technology.

Density

In ASIC terminology, density is the amount of logic that can fit on a chip. The programming overhead of FPGAs dictates that, for the same amount of logic, FPGAs will always be larger and therefore more expensive than MPGAs. However, many MPGA designs are *pad limited* -- the size of the die is dictated not by the number of gates that can be placed in the area, but by the number of I/O pads that surround those gates. Since I/O pads are placed on the periphery, they scale *linearly* with feature size as a result of improved IC manufacturing processes, while the core of the array scales *quadratically*. Since some fraction of the core of the chip is wasted in a pad-limited MPGA design, the use of that area by FPGAs for field programmability would not increase the area of the resulting part. At some point FPGA and MPGA size for a given number of gates will be dictated by the I/O count, so FPGA and MPGA silicon cost and capacity will be the same.

FPGAs have a fundamental advantage in manufacturing efficiency. Since all parts are identical and all test programs are the same, FPGAs require less equipment and less handling to produce the same number of good parts. This efficiency drives down the cost of FPGAs relative to MPGAs for the same die size.

Speed

Despite being smaller and slower than MPGAs, FPGA size and speed are adequate for most applications. Because field programmability slows down the parts, FPGAs will always be slower than the equivalent MPGA for the same application, but there is reason to believe that FPGAs will reduce the gap in performance.

An application typically has a small amount of logic that dictates its overall speed. Dedicated architectural features of FPGAs can eliminate unneeded programmability in speed-critical paths. These features include high-fan-in decoding logic and special-purpose arithmetic logic. Since there is no extra programming circuitry in the path to slow it down, it runs as fast as custom logic. Since it was designed with the intent to produce a high-speed path, it may even run faster than the mask-programmed general-purpose circuitry in an MPGA.

FPGA manufacturers move their FPGAs to faster processes as they become available, allowing a user to speed up the application simply by buying the faster device. In contrast, an MPGA manufacturer will not automatically upgrade an existing MPGA design to a new process, absorbing the expense of the new mask set. The result is that although the MPGA manufacturer has an improved process, the MPGA user cannot access to it with the old design.

Architecture

Most importantly, though, is the relative immaturity of FPGA architectures. MPGAs trace their lineage to IBM's Master Slice of the late 1960s. They have been refined over the years to an efficient structure that merges logic and interconnect, and serves as an easy target for design automation software. In contrast, the first FPGAs were introduced in 1984, and FPGA architectures are still undergoing significant change. More than any other change, these architectural innovations should serve to close the gap with MPGAs as FPGAs mature.

1.6. Designing for FPGAs

The design flow for FPGAs varies among the different types of FPGAs. This section gives an overview of the problems associated with designing with FPGAs and discusses some common issues. Design tools and methodology will be covered in more detail in the following sections, with emphasis on the design flow for the particular FPGA.

FPGA designs can be made with the same tools and techniques used to design MPGAs (figure 1.8). Schematic entry, logic synthesis and equation-based entry are all available from FPGA manufacturers and CAE vendors. Some FPGAs can be programmed with a very low-level device-dependent design editor, for maximum utilization of the device.

FPGA manufacturers and CAE software vendors supply software for optimizing the logic for an FPGA architecture, for mapping the logic into the FPGA efficiently, and for routing the connections through the configurable interconnect. The result of the implementation step is a programming file that can be loaded into the FPGA or used to control the FPGA programmer to customize the part.

The current generation of ASIC design tools was developed to support MPGAs and standard-cell chips. Since MPGAs and standard cell chips are typically made by interconnecting standard gates in custom patterns, the tools retain this bias. This bias is particularly evident in logic optimization and technology mapping, where the software attempts to optimize the design to a gate-like architecture. This optimization may not produce a good result on an FPGA that does not follow the MPGA model. Different FPGA vendors address this problem differently. Some have chosen to address software compatibility as a fundamental issue, others have chosen to adopt a more MPGA-like "synthesis friendly" FPGA architecture. These software and architecture issues will be addressed independently for each architecture.

Design Migration

FPGAs provide cost-effective design and implementation, while MPGAs provide low-cost volume production. A designer can have the best of both worlds by prototyping a design on an FPGA, then switching to an MPGA for production. An FPGA

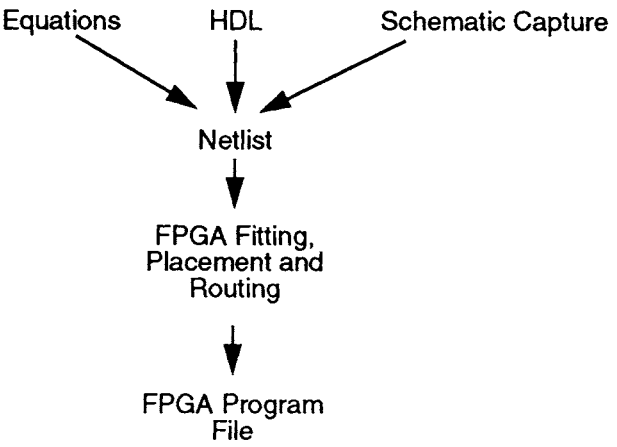


Figure 1.8. FPGA Design System Overview

netlist can be re-targeted to an MPGA by library translation followed by re-verification of all timing paths. The re-verification is difficult if the target MPGA logic does not have the same relative performance as the FPGA logic. Someday, re-targeting software in logic synthesis systems may be able to correct the timing differences, but at present this problem requires manual intervention.

To address this design migration problem, some FPGA vendors offer mask-programmed versions of their field programmable parts with compatible logic structures and delays. The coupling of FPGAs and their mask-programmed equivalent parts give FPGA users the advantages of both implementation methods.

1.7. Outline of Subsequent Chapters

This remainder of this book contains three chapters, each of which addresses a single commercial FPGA architecture and software. Chapter 2 describes the Xilinx SRAM-based architectures, chapter 3 describes the Actel antifuse-based architectures, and chapter 4 describes the Altera EPROM-based architectures.

Each chapter stands as a complete presentation, but the three chapters share a common outline, so readers may make comparisons between the different FPGA architectures and software. The common outline is this:

1. Introduction

This section contains background information on the architecture.

2. Programming Technology

Each of the FPGAs described in this book has a different method of programming. This section describes the manufacturing process and circuit design that provide field programmability.

3. Device Architecture

This section describes the organization of an FPGA, the way it implements logic and the way the logic blocks are connected. This section also addresses density and speed limits of the parts and architectural design trade-offs.

4. Software

Design methodology and flows differ among different FPGA architectures. Architectural decisions may alleviate or emphasize different aspects of the software. This section describes not only the tools needed, but the algorithms those tools use.

5. The Future

Technology trends indicate how the capabilities of these FPGAs will change relative to one another in the future. In addition, this section identifies open areas of research.

6. Design Applications

The advantages and disadvantages of each architecture can be made most apparent with design examples. The example designs in this section show the interaction between the FPGA architecture and the design software.

7. Acknowledgments

8. References

References for each type of FPGA are listed separately for each chapter. A single reference may appear in more than one chapter.

1.8. References

D.G. Reinertsen, "Whodunit? The search for the new-product killers," *Electronic Business*, July 1983.

P.G. Smith, D.G. Reinertsen, *Developing Products in Half the Time*, Van Nostrand Reinhold, New York, 1991.

C. Ebeling, G. Borriello, S.A. Hauck, D. Song, E. Walkup, "TRIPTYCH: A New FPGA Architecture", in *FPGAs*, W. Moore and W. Luk, ed., Abingdon Press, Oxford, UK, 1991.

E.K.F. Lee, P.G. Gulak, "A CMOS Field-Programmable Analog Array", *1991 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, IEEE, 1991.

P. Chow, S.O. Seo, D. Au, T. Choy, B. Fallah, D. Lewis, C. Li, J. Rose, "A 1.2um CMOS FPGA using Cascaded Logic Blocks and Segmented Routing", in *FPGAs*, W. Moore and W. Luk, ed., Abingdon Press, Oxford, UK, 1991.

Chapter 2
SRAM Programmable FPGAs

Steve Trimberger, Xilinx, Inc.

2.1. Introduction

Since their introduction, SRAM-programmable FPGAs have become very popular. Carter [1986], Hsieh [1987, 1990], Kean [1989], Furtek [1990], Hastie [1990], Kawana [1990], Muroga [1991], Ebeling [1991], Chow [1991], Hauck [1992], Hill and Britton [Hill 1992][Britton 1993] and Cliff [1993] have all proposed SRAM-programmable FPGAs.

This chapter focuses on the three Xilinx families of FPGAs [Carter 1986] [Hsieh 1987] [Hsieh 1990] as representatives of the class of SRAM-programmable FPGAs with mature software. These three FPGA families share a common structure: an array of configurable logic blocks surrounded by configurable interconnect. The three families differ in the details of the logic and interconnect structures. Members of a family have identical block and wiring structure, but differ in the size of the array. This chapter begins with an overview of the programming technology, it covers device architectures for the three devices, software and design applications.

2.2. Programming Technology

SRAM Programming

An SRAM-programmable FPGA is programmed by loading *configuration memory cells* from an external source. The configuration memory cells control the logic and interconnect that perform the application function of the FPGA. There is no separate RAM area on the chip, the memory cells are distributed among the logic they control.

The configuration memory is written only once for each application so, unlike commercial static RAM memory chips, high-speed read and write is not important. Stability and density are primary concerns. Figure 2.2.1a shows the CMOS five-transistor memory cell used in Xilinx FPGAs. The Read/Write pass transistor (R/W), is used both to load the cell and to read back the programming. During normal operation it is off, and the cell holds its programming.

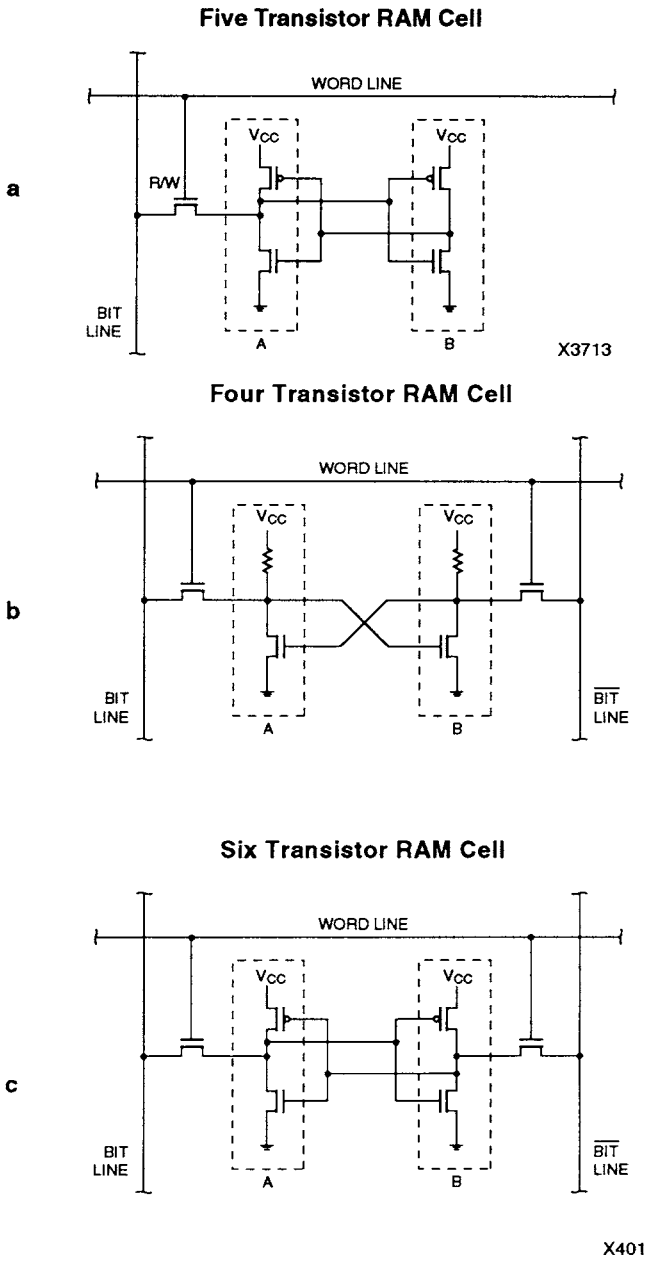


Figure 2.2.1. a) Xilinx Five-Transistor Configuration Memory Cell. b) Four-Transistor Memory Cell. c) Six-Transistor Memory Cell.

The six-transistor memory cell used in CMOS memories uses the data bit in both the true and complement form (figure 2.2.1c), providing fast read and write times at the cost of another transistor. The four-transistor memory cell often used in high-density SRAMs (figure 2.2.1b) has polysilicon resistors instead of the P-channel pullup transistors. These giga-ohm resistive paths increase sensitivity to soft errors such as alpha-particle upsets. The internal signals in the memory cell in figure 2.2.1a are always connected to one of the two power supplies with a low-resistance path, so the cell is very stable. Alpha particle sensitivity tests of the five-transistor memory cell show the expected mean time between failure due to alpha particle upsets to be nearly one million years [Lyons 1985] [Xilinx SMTO 1991].

Advantages and Disadvantages of SRAM Programming

Volatility

SRAM programming has an obvious drawback -- volatility. When the power is turned off, the IC loses its programming, so an SRAM FPGA must be reprogrammed each time power is applied. SRAM FPGAs include logic to sense power-on and to automatically initialize themselves, providing "virtual non-volatility," provided the application can wait the 2ms-30ms required to program the FPGA. This is usually not a problem because system start-up times are usually longer than this. For systems that need active logic during power-up, the initialization time for an SRAM-programmed part must be considered in the power-up sequence.

External Memory

A related disadvantage of SRAM programming is that it requires an external memory for permanent storage of the program. Although multiple FPGAs can share a single external memory, this multiple-chip solution may be inappropriate where board space is crucial. However, many systems already have significant amounts of storage in initialization microcode, bootstrap PROMs or disks. Many users of reprogrammable FPGAs are able to share the permanent memory used to initialize other components of the system.

Reprogrammability

The disadvantage of volatility provides the advantage of reprogrammability. Reprogrammability makes SRAM-programmable FPGAs ideal for prototype development. Since the FPGA can be reprogrammed without cost, a designer can load a design into the part, try it at-speed in the system and debug the design. If necessary, a modified version of the design can be loaded into the FPGA and tried in the system without removing the chip. One-time-programmable FPGAs require that the defective prototype part be removed and a new part programmed and inserted into the system to make the change.

A reprogrammable FPGA can be time-shared to replace logic amounting to many times its maximum capacity. If a system can be divided into pieces that are not required simultaneously, the pieces can be designed into separate configurations of the FPGA, and the FPGA time-shared between them by repeated reprogramming.

A common use of reprogrammability is for board-level test. As part of the test sequence, the FPGA is configured as a test generator/checker to verify the board or components on the board. After board-level test is finished, the FPGA is re-configured with the application logic. This design saves the additional hardware that would have been needed for system-level test. Other uses allow a system to interface to multiple different external devices, selectable by the programming of the FPGA.

A system built with reprogrammable logic can be updated after delivery to a customer by modifying the programming of the FPGA. The new programming bitstream can come from a new PROM or from memory in some other part of the system. This feature has been used for soft-hardware field upgrades, replacing the FPGA programming from a floppy disk or even from a remote computer over a modem. Applications of reprogramming are discussed in more detail in the Applications section of this chapter.

Quality

Indirectly, reprogrammability leads to very high quality parts because each part can be fully tested at the factory without destroying it. Every programming point and every path is tested. SRAM FPGA tests cover all the typical stuck-at faults as well as many other pattern faults that ASIC test generators might miss. These faults include stuck-open faults and bridging faults that result from layout proximity. SRAM FPGAs can also be tested for speed and binned accordingly, so users can choose only the fastest parts for performance-critical applications.

Programming yield for SRAM FPGAs is always 100%. There is no separate programming step, and no removal and re-insertion for programming. The handling associated with programming antifuse and EPROM-programmed devices damages the package pins, causing problems for board assembly.

Process Leadership

The SRAM process used to build FPGAs is the same CMOS process used to make ASICs and is very similar to the process used for CMOS memories, so SRAM-programmable FPGAs are among the first logic products to take advantage of process improvements driven by semiconductor memories. Since improved processes are both denser and faster than older ones, those advantages apply to SRAM-programmed FPGAs as well.

Low Power

SRAM-based FPGAs implement logic in static gates, so SRAM programmable FPGAs have low power consumption even for very large amounts of logic and have zero standby current. In contrast, EPLD-style FPGAs have passive pullup and sense-amplifier circuitry that leads to prohibitively-high static power dissipation for high-capacity or high-speed parts.

2.3. Device Architecture

All three Xilinx FPGA families consist of an array of Configurable Logic Blocks (CLBs) embedded in a configurable interconnect structure and surrounded by configurable I/O blocks (figure 2.3.1). Each family architecture was driven by a different set of assumptions and a different model of use. These different design pressures led to different block, I/O and wiring architectures, as well as other unique features of each FPGA. Family members differ in their number of blocks and I/Os, with CLB array sizes ranging from 8x8 to 24x24 blocks. These chips support designs in excess of ten thousand gates, with system clock speeds in the tens of megahertz.

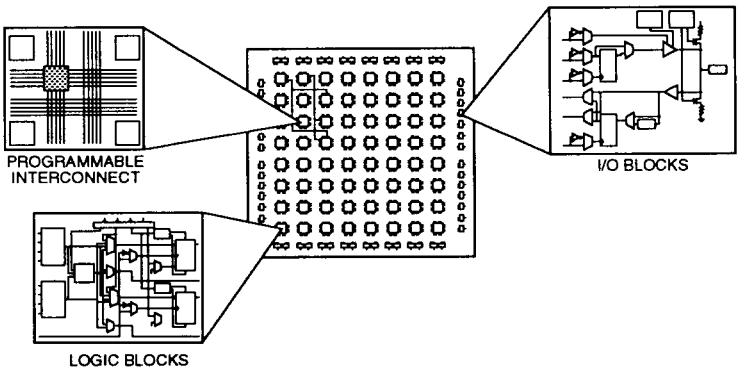


Figure 2.3.1. Island-Style SRAM-Programmable FPGA Architecture.

Simple SRAM-Programmable FPGA Architecture

This section contains a bottom-up description of a simple SRAM-programmable FPGA followed by a discussion of architectural trade-offs. Rather than try to describe a commercial FPGA architecture, the intent is to give the reader an understanding of the underlying programming technology and design methodology without the detail of a particular implementation. Following sections describe commercial architectures and how they address the design trade-offs.

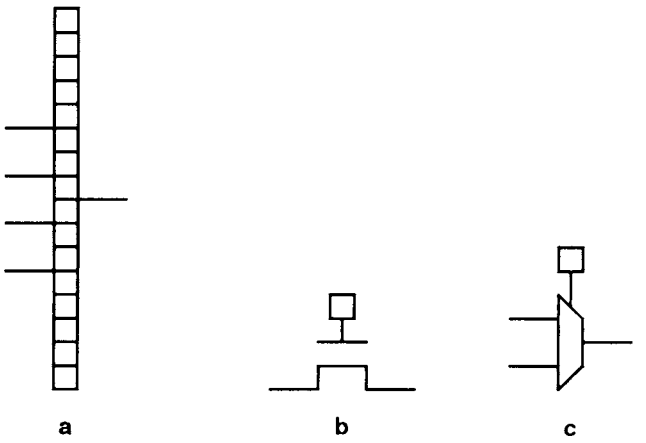


Figure 2.3.2. Three Important Pieces. a) Lookup Table. b) Pip. c) Multiplexer Controlled by a Configuration Memory Cell.

Building Blocks

Lookup Table. Figure 2.3.2a shows a four-input *lookup table* (abbreviated *LUT*) or *function generator*, a basic unit of configurable logic. A lookup table implements combinational logic as a 2^nx 1 memory composed of configuration memory cells. The memory is used as a lookup table, addressed by the n inputs. A lookup table can implement any of the 2^{2^n} functions of its inputs. When the FPGA is programmed, the memory is loaded with the bit pattern corresponding to the truth table of the function [Horowitz and Hill 1989]. For example, if all configuration bits in the 16-bit lookup table in figure 2.3.2a are 0 except the high-order bit (address 15), the output of the lookup table will be zero unless all inputs are high (binary address 1111): a 4-input AND. To implement functions of fewer inputs, the unused inputs are held low or the subset of the truth table entries is duplicated to make the output the same regardless of the value of the unused input.

The inputs to the lookup table are logically equivalent -- any signal can be connected to any input pin of the lookup table. Changing the pin to which a signal is connected requires a straightforward rearrangement of the bits in the lookup table. All functions of a lookup table have the same timing: the access time of the memory. Placement and routing software can take advantage of the logically-equivalent pins, choosing which input connects to which pin to optimize routing.

Programmable Interconnect Point. The second building block is called a *programmable interconnect point*, (*pip*). Some authors use the term "configurable interconnect point" (cip). Pips control the connection of wiring segments in the programmable interconnect. The pip, shown schematically in figure 2.3.2b, is a pass

transistor controlled by a configuration memory cell. Wire segments on each side of the transistor are connected or not depending on the value in the memory cell. A pip is the basic unit of configurable wiring.

The pip switches the signal with a pass transistor, not a full CMOS transmission gate. The transmission gate requires three more transistors: one for the transmission gate and two for the inverter that negates the control signal. A full CMOS transmission gate requires two transistors connected to each wiring segment for each pip, adding diffusion capacitance to the segment, slowing down the interconnect. The drawback of the single pass transistor design is *threshold drop*: signals in the interconnect are not pulled all the way up to the high voltage, so all logic inputs must either dissipate power due to lowered high-voltage on the P-channel pullup, drive the P-channel destination with a restored signal, or include a CMOS P-channel pullup with a lower threshold to restore the signal logic.

Multiplexer. The third building block is a multiplexer controlled by a configuration memory cell (figure 2.3.2c). The multiplexer is a special-case, one-directional routing structure. It may be of any width, with more configuration bits for wider multiplexers. Switches built with multiplexers reduce the number of memory cells required for controlling the switch, giving an area savings for large switches.

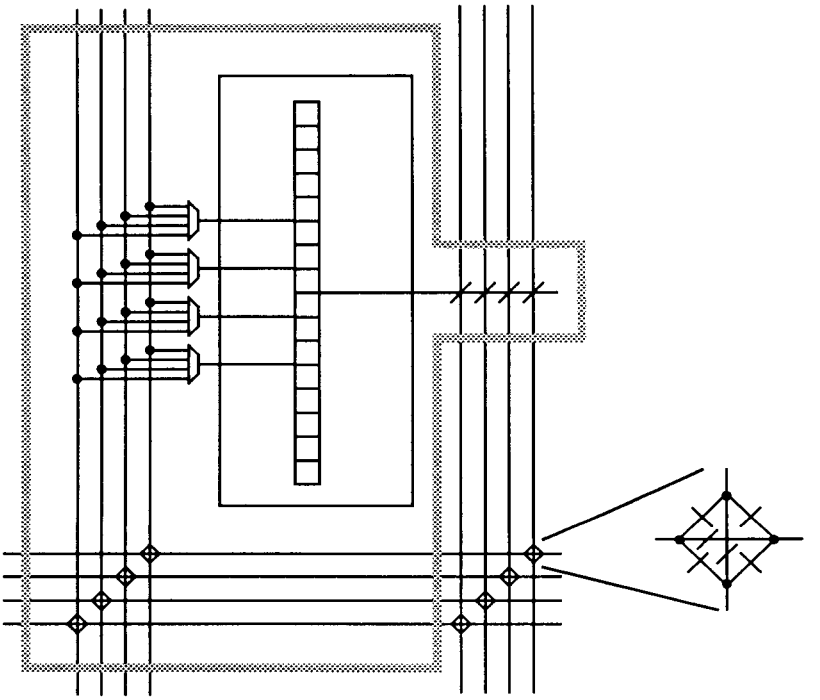


Figure 2.3.3. An FPGA Tile.

The Logic Block

Figure 2.3.3 shows the building blocks from figure 2.3.2 combined into a single *tile*, shown outlined, composed of a *configurable logic block* (*CLB*) and wiring. The CLB in figure 2.3.3 contains a single four-input lookup table. The CLB is surrounded by *wiring channels*. Each wiring channel contains several wiring *segments*. Segments in the left side channel have multiplexers on them to make connections to the CLB. The CLB output is connected to the wiring segments at the right through pips. For simplicity, pips are shown as diagonal lines at the intersections of the segments in the channels and between segments in the switchbox. If the block is unused, the output pips are all turned off, so the block does not drive its signal onto any segment, and the segment may be used to connect other signals on the chip.

Where the wiring channels intersect, their connections are made by a pattern of pips in a *switchbox*. For routability considerations, a crossbar switch in the switchbox is ideal, but a full crossbar switch connecting n wires requires n^2 pips, so it is prohibitively large. In practice, the switchbox contains only a few of the possible connections between segments in intersecting channels. A signal may be routed through pips in switchboxes to other blocks or back to the same block. Each segment in the switchbox in figure 2.3.3 has three pips, one to a segment on each of the other three sides of the switchbox. The number and pattern of connections in the switchbox has a significant effect on the routability and performance of the FPGA, as will be discussed later.

The tile in figure 2.3.3 can implement sequential logic as well as combinational logic. To build a latch, the lookup table is configured as shown in figure 2.3.4, and the output is routed back to the input. The first and second inputs to the lookup table are configured for the reset and set signals. A clocked latch can use the fourth input for the clock. Latches can be combined to implement other kinds of sequential elements, such as D-type or T-type flip flops.

The Chip

Figure 2.3.5 shows a chip composed by building an array of CLB tiles, then surrounding the array with configurable *I/O Blocks* (*IOBs*). An IOB allows signals to be driven off-chip or optionally brought onto the chip onto interconnect segments. The IOB may perform other functions, such as three-state outputs and registering incoming or out-going signals.

Different family members in a common architecture are constructed by assembling different-sized arrays of the tile and surrounding that array with different numbers of IOBs.

The FPGA must include circuitry to load the *configuration bitstream* or *program* into configuration memory where the memory cells produce logic functions in lookup tables and control interconnect in pips and multiplexers. Typically, the bits in the

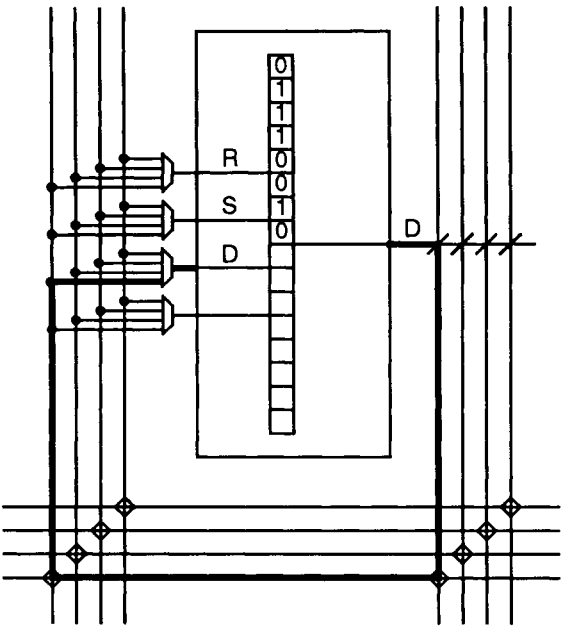


Figure 2.3.4. The Tile Configured as a Latch.

program are loaded serially, and the programming of the FPGA can be considered to be a single, long programming word.

Design Trade-offs

Density and Speed

Density is the amount of usable logic per unit of chip area. Density has two components: the size of the structures that implement logic and the logic capacity of those structures. The size of components in an SRAM FPGA correlates fairly well with the number of memory cells required to build and control them. The capacity of the structures depends on how well software can map logic from typical designs into the structures the FPGA implements. More general structures are preferred, since more logic can be put into them and they require simpler design automation software. However, more general connections usually require more controlling logic, so they take more area.

The area dedicated to interconnect does not contribute to logic capacity, so an FPGA with minimal interconnect may appear to have excellent logic density. However, much of the logic will be inaccessible to designs built on the FPGA because of the

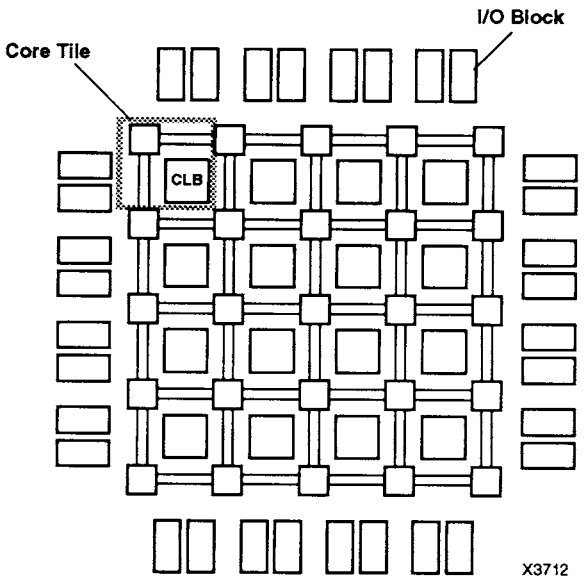


Figure 2.3.5. An FPGA as an Array of Tiles.

inadequate wiring resources. As a result, the wiring-poor FPGA will have comparatively little *usable* logic.

The *speed* of an FPGA is a measure of the delay required to implement a function and to propagate signals to neighboring functions. The speed of the logic part of the block is the sum of the delays from the input selection multiplexers, the lookup table and the output drivers. For complex blocks, the delays of different paths through the block may be different.

The fraction of delay incurred due to interconnect in an FPGA is significantly greater than that in an MPGA because configurable interconnect is inherently much slower than mask-programmed interconnect. In an MPGA, wiring is implemented on metal runs that have low capacitance and low resistance. While an SRAM-programmable FPGA also uses low-resistance metal for interconnect, a signal passes through a transistor at each pip (figure 2.3.6). The channel resistance of the transistor that makes up the pip and the capacitance due the source/drain diffusions of all pips on the segment determine the speed that signals propagate through the interconnect. In figure 2.3.6, a signal from output A to input B sees the resistance of the pip after the output buffer, the loading on segment s4, the resistance of the pip to segment s1, another segment load, and finally the input multiplexer. The equivalent schematic of the path is shown in figure 2.3.6b, where C_s is the segment loading capacitance, equal to the capacitance of all the pips on the segment plus the capacitance of the metal

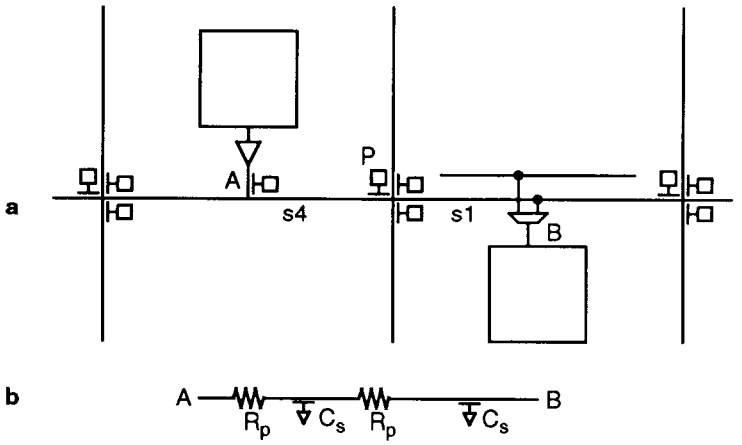


Figure 2.3.6. Interconnect Segment Detail. a) Architecture-Level. b) Electrical Equivalent.

segment, and R_p is the resistance of a pip. Signals that travel longer distances have additional resistance and capacitance stages.

It is the task of both the architecture and the software to limit signal delay and to provide buffering for long wires. The architecture may provide buffering in the interconnect, or the routing software may be required to route signals that need buffering through unused CLBs. Both options have been employed in SRAM FPGAs, and they will be discussed with the individual FPGA architectures.

Speed and density must be traded off against one another. Faster output buffers are larger. CLBs with a large number of inputs can make high-fanin logic faster, but each input multiplexer costs area and hence reduces density.

Size versus Routability

An architecture with more pips provides more options to the router, simplifying routing, but each pip includes a memory cell, adding measurably to the size of the FPGA. Rose and Brown [1991] investigated the routability of lookup-table-based FPGAs under differing numbers of pips in the switchbox and in the routing channel. They placed and routed a set of designs on a simple FPGA architecture, varying the number of tracks in the channels, the number of pips from CLBs to tracks and the number of pips between tracks in switch boxes. They report the number of tracks in excess of the channel density required to route the set of designs over a range of pip distributions. Their results show that, for a high probability of complete routing, the architecture must have good connectivity in the channel, with at least seventy percent

of the channel segments connectable to the CLB; but connections in the switchbox may be sparse, with as few as three or four connections from each segment in a switchbox.

Speed versus Routability

Every segment of configurable interconnect is capacitively loaded by the pass transistors of all potential connections that can be made from that segment, as shown in figure 2.3.6. When a signal is routed through a pass transistor to another segment, the net includes the resistance of the transistor. The delay of a net is the time required to charge or discharge the distributed capacitance of all segments through the resistances of all pips and the CLB output driver. The additional delay on a net from adding a routing segment grows quadratically with the number of pips on the net, due to the distributed resistance of the pips and the capacitance of the segments. This is the same delay behavior observed in a lossy transmission line. To improve performance, the resistance of a pip can be reduced by making the pass transistor wider. However, wider transistors load the segment with more capacitance. In practice, interconnect delays are about the same as the block delays, but for pathological cases, they can be much larger.

Because signal delay is more dependent on the number of pips through which a signal is routed than on the distance covered, FPGAs typically have interconnect wiring segments in a variety of lengths. This can be done by replacing some "straight-ahead" pips (such as pip P in figure 2.3.6a) with a wired metal connections. This saves the area of the pips, and eliminates the series resistance of the pips along the wiring track. The capacitance of a segment is larger because the one segment still has all the branching pips. By eliminating many switchbox pips, we can make a channel with wiring segments of many different lengths. The result has been called a "segmented routing channel" [Greene 1990].

If a signal travels the full length of a multiple-length segment in a segmented routing channel, it reaches its destination in less time than it would if it were routed on single-length lines. However, if the signal is not required to travel the whole length of the segment, the delay may be greater due to the larger capacitance of the longer segment. Further, if much of the long segmented routes are wasted, the overall routability of the chip is degraded. Therefore segmented channels require more tracks than channels with only single-length lines.

Another architectural option is to include direct connections from the output of a block to the inputs of nearest-neighbor blocks, bypassing both the series resistance of the pips and the capacitive loading of the segments. These connections provide high-speed paths for simple connections. This technique has also been called "cascade." Finally, an FPGA may also have longer-distance connections or special facilities for long-distance or high-fanout connections.

Block Size and Structure

A lookup table is not the only way to implement logic. For example, instead of a lookup table, we can implement logic with a non-programmable four-input NAND gate. The NAND gate requires a fraction of the transistors of the lookup table and runs somewhat faster, although several NAND gates may be required to implement a function that would fit into a single lookup table. For more generality, one could implement a few different functions of the inputs in a logic block and select the one desired function with a programmable multiplexer. This solution still requires some control circuitry: memory cells to control the selection of the function of the block from the set of implemented functions.

Simple blocks, such as NAND gates or two-input lookup tables, are conceptually elegant, and design software can usually use them efficiently. Further, a large block may not always be fully utilized, leading to lower logic capacity.

When considering size and performance trade-offs, one must consider not only the logic element, but also the interconnect segments, pips and multiplexers required to connect the elements. Because the lookup table can implement any function of its inputs, a single lookup table replaces several simple gates as well as the delay and area-intensive interconnect between them. In practice, implementing logic in a lookup table produces a smaller, faster FPGA because the lookup table efficiently combines versatile logic implementation with the programming overhead of the memory cells.

This trade-off can be illustrated with a consideration of lookup table widths. A four-input lookup table has sixteen memory cells, a three-input lookup table has only eight. If the logic being implemented in 4-input lookup tables breaks naturally into three-input gates, half of every 4-input lookup table area will be wasted, leading to an inefficient implementation on the FPGA. On the other hand, if the logic tends to break into four-input pieces and we implement it in 3-input lookup tables, each 4-input function will need three three-input lookup tables plus the associated interconnect. The resulting logic is not only slower, but also larger than the four-input version.

Rose, et. al. [1990] investigated a range of lookup table sizes and their effect on the overall chip area, including both CLB area and routing area. They mapped logic from a number of designs into lookup tables and estimated the size of the chips that would have been required to implement those designs. Their estimate used total memory cell count and wiring cost estimates. They performed their analysis for a range of lookup table sizes and programming cell sizes. Their results show that a three-input or four-input lookup table gives the best density for a wide range of programming cell sizes. Larger lookup table sizes were preferred for a high-speed architecture.

This type of trade-off analysis applies to sequential logic elements as well as combinational logic. Figure 2.3.7 shows a dedicated flip-flop in the logic block with a control multiplexer to select either the lookup table or the flip-flop output. The dedicated flip-flop, when used, saves two lookup tables and their associated wiring.

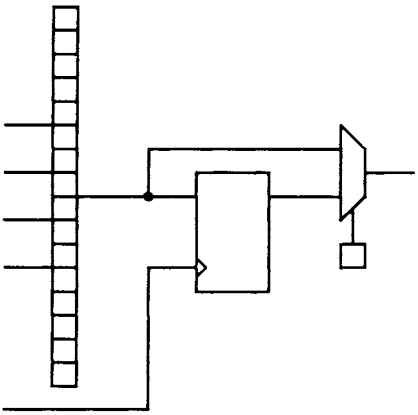


Figure 2.3.7. A CLB with Registered Output.

Although the flip-flop area is wasted when the combinational output is desired, the area savings over all designs for using the flip-flop exceeds the waste when it is unused [Rose 1990]. In addition, the dedicated flip-flop provides predictable setup time, hold time and metastability parameters. If a latch is built from a lookup table and interconnect, as shown in figure 2.3.4, the D connection from the output to its non-feedback destination loads the feedback path, making the timing of the latch dependent on its wiring. Buffering the signal requires an additional lookup table, doubling the size of the latch.

The determination of the desirability of a dedicated function in the FPGA architecture can be generalized into a pair of design rules.

1. A dedicated function improves the density of an FPGA if, over all designs, the area wasted when the function is unused is less than the total area saved when it is used.

2. A dedicated function improves the performance of an FPGA if, over all designs, the delay along the critical path added by the feature is less than the delay reduction along the critical path due to the feature.

The value of a dedicated feature is also dependent on the ability of design software to use it and on the "general applicability" of the feature. A feature that is unused by design software can not improve the quality of implementation. A feature that improves the FPGA for some designs and degrades it for others will be of limited value.

Capacity Estimation

FPGAs have three kinds of resources: logic, I/O and routing. To determine if a design fits into a particular FPGA, the design must fit within all three resource limits. The difficulty of this estimation is a function of the architecture and of the software used for mapping the logic into the FPGA. FPGA logic and interconnect capacity are difficult to estimate. Traditional measures of gate count and product terms are not accurate estimates of lookup-table capacity. Two designs that appear to be of equal size in terms of MPGA gate count or number of PLD product terms may use CLBs with different efficiency, requiring very different numbers of CLBs. Logic optimization algorithms may also significantly change the size and performance of the design.

Complex blocks implement complex functions efficiently, but when the function to be implemented does not fit into the block efficiently, some fraction of the block is unusable, and is wasted. The wasted fractions of blocks cause a gap between the peak capacity of the FPGA and the capacity in a given application. An accurate capacity and performance estimate requires that the design be mapped into the FPGA. Fortunately, fast mapping heuristics can give a good estimate of logic capacity.

Routing requirements are more difficult to estimate. The problem of statistical wirability estimation has been addressed by Heller [1978], Donath [1979] and ElGamal [1981], but the techniques and results are not accurate enough for capacity estimation. MPGA designs address this problem by providing significantly more interconnect than is needed by most designs. This solution in impractical in FPGAs because unused FPGA interconnect degrades performance and density too severely. FPGA design systems include high-speed placement and routing for routability estimation and timing-driven routing to meet delay requirements.

The Xilinx XC2000 Architecture

The Xilinx XC2000 family [Carter 1986] was the first commercially-available FPGA. Introduced in 1985 and still used today, the XC2000 architecture was developed without supporting software to verify logic density or capacity. The block structure was derived from a general understanding of the way logic is decomposed in typical applications and from manual implementations of existing MPGA designs. A crucial concern in the design of the XC2000 family was to build a chip that was small enough to be manufacturable with the IC process available at the time. Therefore, a smaller, slower cell was preferred to a larger, faster cell.

The XC2000 CLB

The XC2000 CLB combinational logic section (figure 2.3.8) consists of two three-input lookup tables producing the F and G signals. The two lookup table outputs can be multiplexed together to produce any function of the four inputs on both outputs. Hill [1991] describes this arrangement as a single four-input lookup table with two

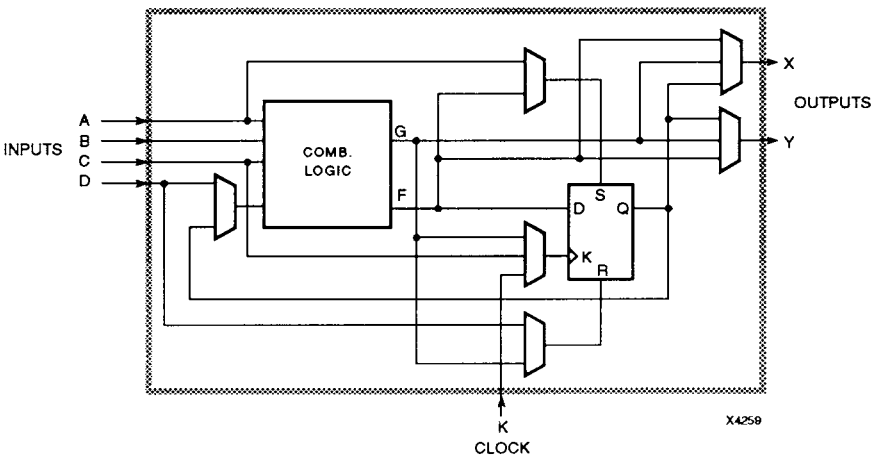


Figure 2.3.8. The XC2000 CLB.

outputs. The CLB includes a single storage element that can be configured as an edge-sensitive D-type flip-flop or as a level sensitive D-type latch. The data input to the storage element comes from the output of the F lookup table. The clock input can come from the G lookup table, the C input to the CLB or from a separate clock input, K.

Either of the CLB outputs can be configured to be the result of the F lookup table, the G lookup table or the sequential result, Q. The output of the flip-flop can be recycled directly to the inputs of the lookup table, providing an efficient method of generating state machines and counters.

The designers of XC2000 architecture determined that four-input lookup tables efficiently implemented most logic and that a dedicated flip-flop was generally applicable. This implied an architecture much like figure 2.3.7. However, they did not want to lose half the lookup table for related functions of three or fewer inputs, such as the sum and carry in a full adder. Therefore the XC2000 CLB has two outputs. For a full adder, it can implement the sum in one lookup table and the carry in the other. The sum can be stored in the CLB flip-flop to build an accumulator efficiently.

The XC2000 IO Block

Figure 2.3.9 shows the XC2000 I/O block structure. All chip outputs can be three-stated and bidirectional. The three-state control can be fixed in the configuration bitstream to make the block input-only or output-only, or it can come from a signal in the FPGA interconnect, so on-chip logic can control the direction of the I/O pads. The input signal can be latched in the IOB, reducing hold times for latched inputs that would otherwise have to be wired to a CLB flip-flop.

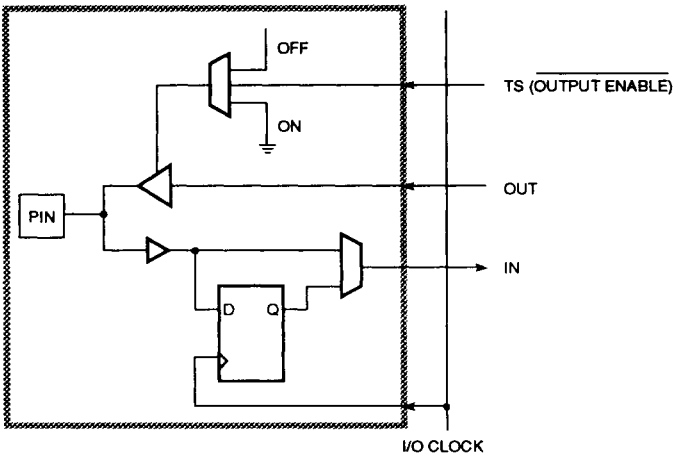


Figure 2.3.9. The XC2000 IOB.

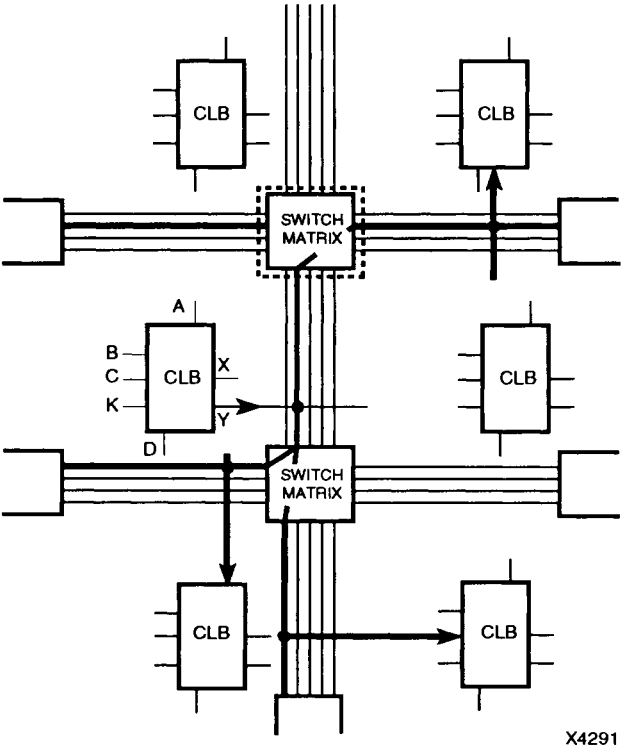


Figure 2.3.10. XC2000 Interconnect Structure.

Wiring Architecture

The XC2000 includes four horizontal and five vertical *general-purpose* interconnect segments between switchboxes in the array (figure 2.3.10). The switchbox pips connect the segments in pairs, with the fifth vertical segment making connections to some of the adjoining segments (figure 2.3.11). The segments can be grouped into channels in which each segment is part of a track.

A net routed on general interconnect shows a distributed RC delay. Sizing all drivers to overcome the worst-case load across the chip is impractical. Instead, the XC2000 provides *repowering buffers* in the interconnect to speed up long-distance connections. The array of tiles is divided into nine sections in a grid arrangement (figure 2.3.12) with general interconnect signals re-powered every time the signal crosses from one section into the next. Local signals within a grid section are not repowered.

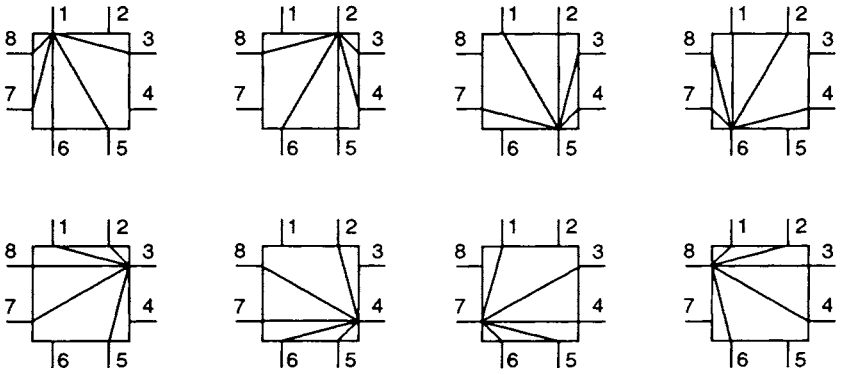


Figure 2.3.11. XC2000 Switchbox Connections.

The XC2000 wiring includes *direct connections* to horizontally and vertically adjacent blocks. These connections are at the ends of the bold lines in figure 2.3.13. A direct connection provides a high-speed dedicated interconnect path to two adjacent CLBs through a single multiplexer input, avoiding the general-purpose interconnect. It is useful for those cases where a signal has a single speed-critical destination and that destination can be placed next to the source. If this ideal placement cannot be made, the connection can still be made by routing through general-purpose interconnect.

The XC2000 wiring includes two vertical *long lines* and one horizontal *long line*. A long line is a single metal segment that spans the entire width or height of the array of tiles, bypassing all switchboxes. Signals are buffered onto the long lines, so that they

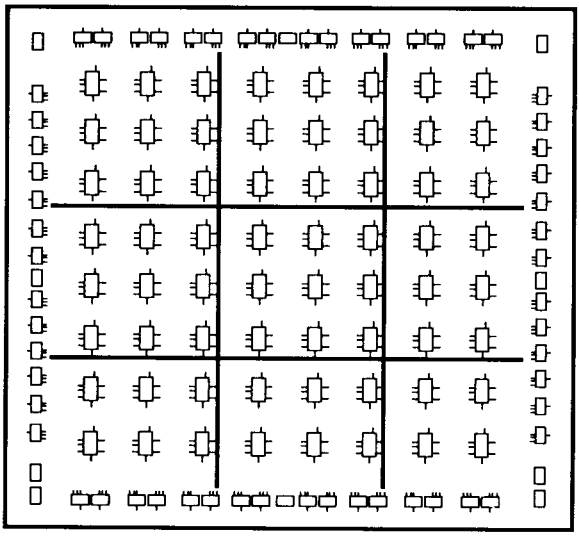


Figure 2.3.12. XC2000 Repowering Buffer Pattern.

can be distributed over a long distance and to a large number of destinations quickly and with low skew.

To further support high-fanout, low-skew signals, the XC2000 FPGA includes two high-drive buffers with dedicated *global* interconnect to all CLBs in the whole chip. These buffers are intended to be used for clocks or other synchronizing signals. This global interconnect reduces the need for a user of the FPGA to do complicated special-purpose clock routing or simulation, as is required in an MPGA to guarantee low-skew clocking paths.

Block-to-Interconnect Connections

The connections to the CLB are distributed around the four sides of the block, with inputs on the bottom, left and top; and outputs on the right (figure 2.3.14). This asymmetry encourages the implementation of a design where signals flow from left to right and top to bottom across the chip. Input multiplexers in the figure, shown as boxes, appear as independent pips to improve the clarity of the drawing. Because equivalent lookup table inputs come from different channels, the router has the flexibility to swap pins.

The outputs of the CLB can connect to only half the segments in the channel. Since CLB inputs can come from any segment, and since signals can switch tracks in the switchboxes, the limited number of output connections is not a serious routability

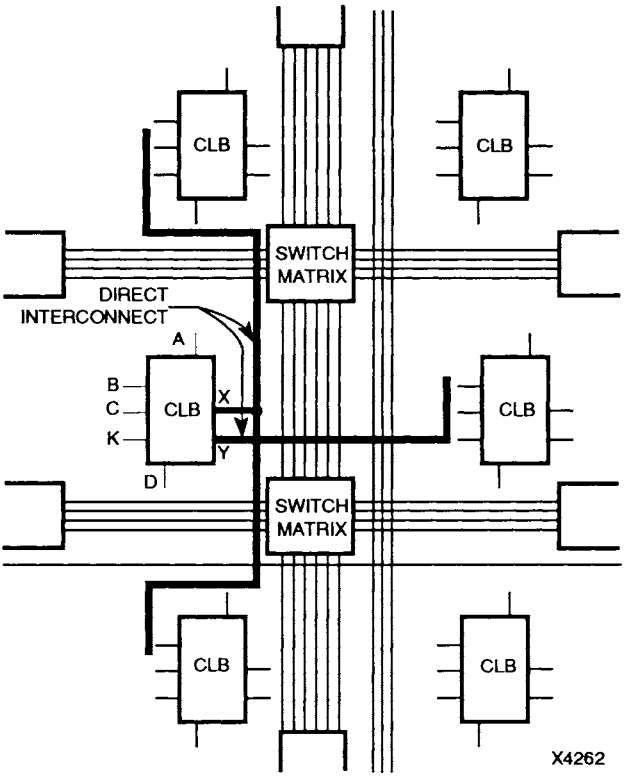


Figure 2.3.13. XC2000 Direct Interconnect.

problem. During routing, if an output must connect to a segment to which it does not connect directly, the router can swap the outputs of the CLB. Output swapping involves exchanging the lookup table contents of F and G and corresponding input selection multiplexers, thereby swapping the signals on the output pins.

XC2000 Family Members

The XC2000 family consists of two members, the XC2064 and the XC2018 (see table). The maximum gate capacity number is a rough estimate that assumes full utilization of all CLBs on the chip. No one design will use all the logic capacity of every CLB, so the typical gate capacity is lower than the maximum.

The XC2000 architecture has not changed since 1985, but improvements in the design and processing have improved the performance of the parts significantly over that period. XC2000 FPGAs operate at flip-flop toggle frequencies up to 100 MHz. The toggle frequency is the speed path from a CLB flip-flop, back through the lookup table and into the flip-flop. It is a measurable maximum frequency at which the part is

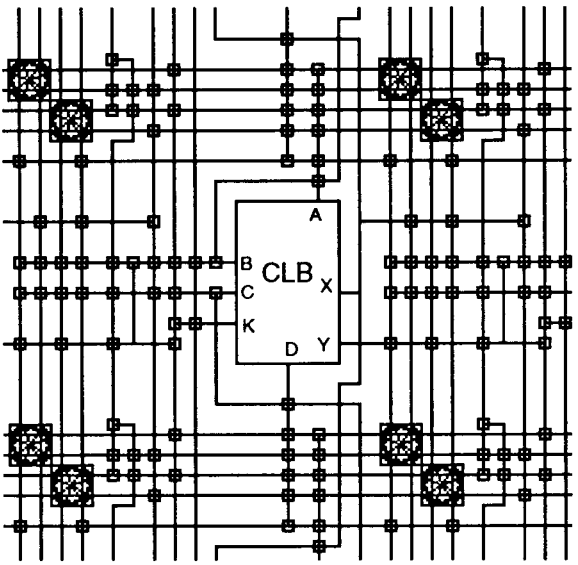


Figure 2.3.14. XC2000 Block-to-Interconnect Connections.

Table 1: XC2000 Family Members

Member	CLB Array Size	IOs	Gate Capacity	
			max	typical
XC2064	8x8	58	1200	800
XC2018	10x10	74	1800	1200

guaranteed to operate. More complex functions require inputs from other sources, and since those inputs must come through the interconnect, they will be slower. Typical worst-case system frequencies depend on the number of lookup tables and on the wiring in the critical speed path. Typical system clock speeds for the XC2000 are about one-third to one-fourth the maximum toggle frequency.

The Xilinx XC3000 Architecture

The XC3000 family of FPGAs is the most widely used FPGA family, with offerings from Xilinx, Seiko and AT&T. The XC3000 architecture includes enhancements to the XC2000 architecture to improve performance, density and usability. The result is a powerful collection of logic capability in a cost-effective package. Key to the

popularity of the XC3000 family is the capacity of the parts, reaching well into the thousands of gates.

The XC3000 architecture was developed with manual tools for design implementation, and the architecture shows a bias toward manual design. This bias is exhibited primarily in the patterns of connectivity in the interconnect. These patterns are usable by a human designer, but provide a difficult set of constraints for software design automation.

The XC3000 CLB

The XC3000 CLB (figure 2.3.15) is substantially larger than the XC2000 CLB. Each of the lookup tables has four inputs rather than three, hence requires sixteen bits of configuration memory rather than eight. The two lookup tables can be combined with a multiplexer to produce any function of five inputs and some functions of up to seven inputs. The wider functions of the CLB extract only a slight delay penalty, and they allow the XC3000 architecture to implement faster logic, since speed critical paths can be implemented with fewer CLBs in series.

The XC3000 CLB has two flip-flops, to ensure that all combinational logic can be followed by a pipelining flip-flop. As in the XC2000, the flip-flop outputs can be fed back internally in the CLB to serve as inputs to the lookup tables. This register-rich

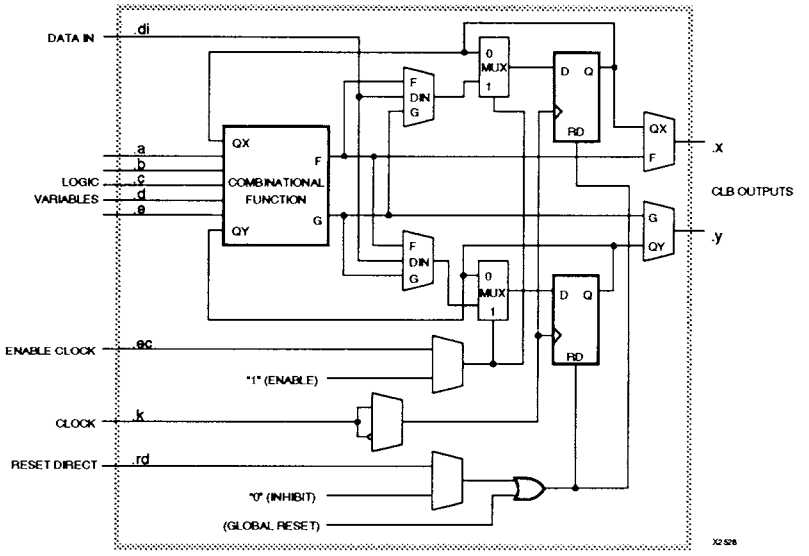


Figure 2.3.15. The XC3000 CLB.

cell lets the XC3000 implement state-intensive applications and heavily pipelined designs efficiently.

The XC3000 IO Block

The XC3000 IOB, shown in figure 2.3.16, is significantly more complex than the XC2000 IOB. The most important addition is a flip-flop in the output path. By registering data in the IOB, the clock-to-out time does not include interconnect delays. The result is a fast, predictable clocked output. The XC3000 IOB also includes a programmable pullup, optional output inversion and selectable slew rate. A slower I/O slew rate for low-speed signals reduces power surges, simplifying board-level design.

Inputs from the pad can be brought into the interior of the chip either directly or registered or both. By allowing both, the I/O block can de-multiplex external signals such as address/data busses, storing the address in the IO flip-flop and feeding the data directly into the wiring.

Wiring Architecture

The XC3000 interconnect structure shown in figure 2.3.17 has five general interconnect lines both vertically and horizontally. In addition, each CLB has direct connections to adjacent CLBs both vertically and horizontally.

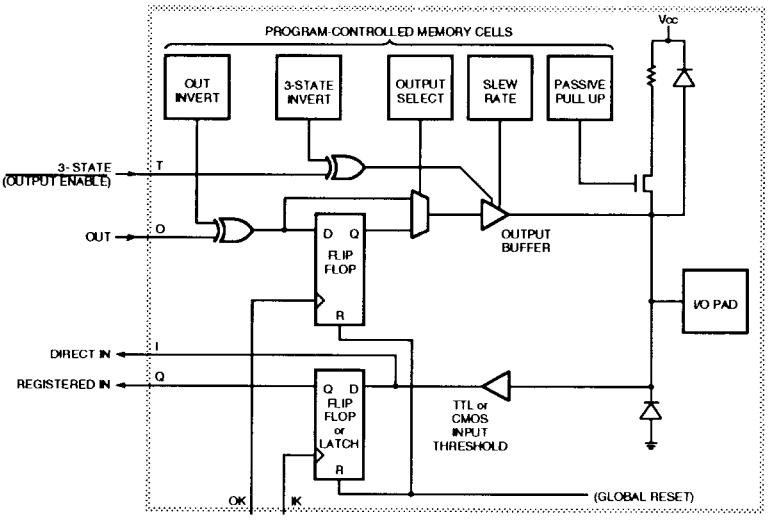
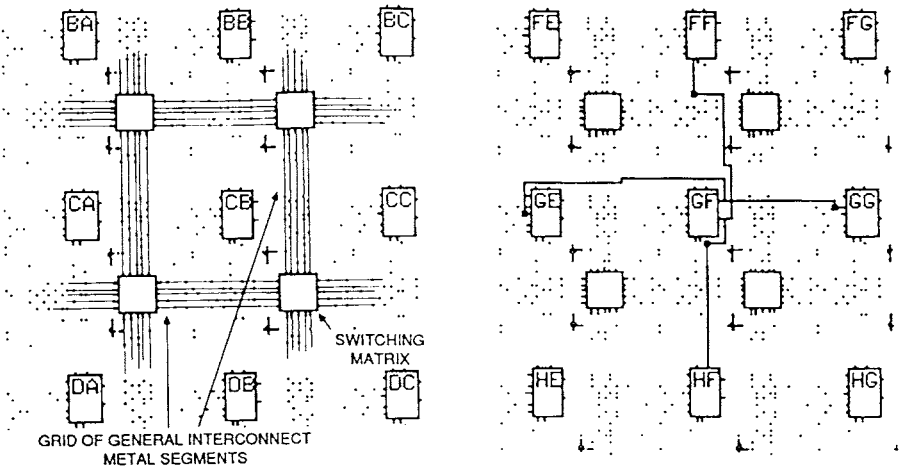
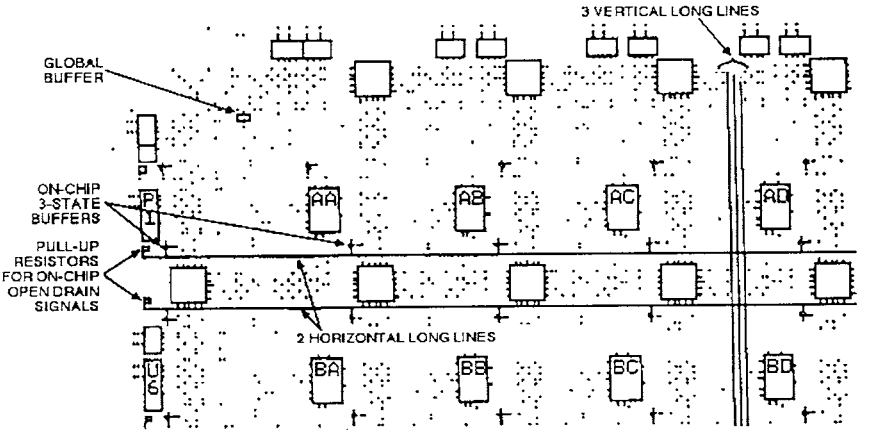


Figure 2.3.16. The XC3000 IOB.



a



b

Figure 2.3.17. XC3000 Wiring.

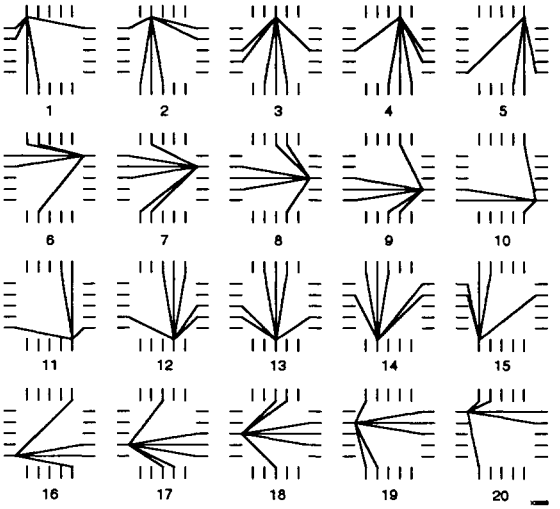


Figure 2.3.17 continued. XC3000 Wiring.

The switchbox connections in the XC3000 are more uniform across the segments, with the interconnect pattern including all wiring tracks. This additional flexibility gives the XC3000 router the ability to avoid blockages by moving signals from one track to another as they are routed across the chip.

Since the XC3000 tile was intended to be built into larger arrays than the XC2000, it includes more long interconnect. There are three vertical and two horizontal long lines. As in the XC2000, signals are always buffered when driving long lines.

Two of the horizontal long lines are driven by three-state buffers, distributed along the long line, one set per CLB. These buffers give users the ability to build on-chip busses. The busses allow implementation of datapaths without the wiring congestion of the wide multiplexers that would be required if three-state busses were not available. Each horizontal long line includes an optional pullup resistor to pull the long line high if none of the buffers is driving it. Configuring the three-state buffers open-drain, connecting the same signal to the input and control of the three-state buffer, and activating the pullup resistor creates a distributed wired-AND.

The three-state buffers on the long line are simple to implement because they are made up of programming features that are already present in the chip. All long line buffers are necessarily three state buffers, because only one signal drives the long line

in standard use. In the XC2000 the three-state control is connected to a programming cell only. The XC3000 includes an optional alternate connection to the three-state driver that comes from the interconnect (figure 2.3.18).

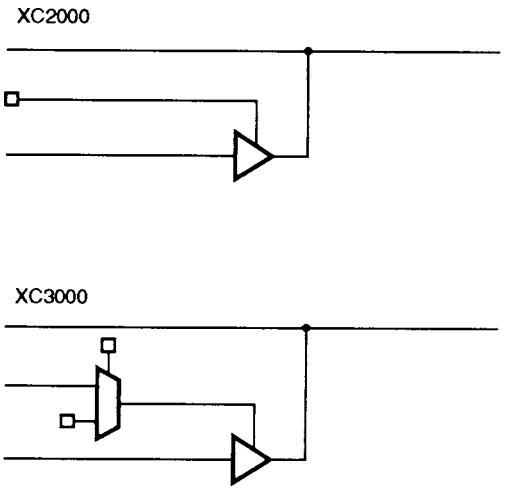


Figure 2.3.18. XC2000 Long Line Buffer and XC3000 Three-State Long Line Buffer Circuitry.

Experience with the XC2000 repowering buffers showed them to be difficult to use effectively. Signals should be buffered after they pass through several switchboxes, but the XC2000 architecture re-drives signals whenever they cross the grid boundary, regardless of the distance they have travelled. A signal to be connected to an adjacent CLB is buffered if it happens to fall at the edge of the grid pattern, whereas a signal that travels a long distance wholly within one of the squares of the grid will not be buffered at all. In both cases, the signal is slower than it would be with intelligent buffering. The XC3000 solves this problem with a combination of hardware and software. The XC3000 interconnect contains redrive buffers scattered among wiring segments. The redrive buffers can be configured to drive a signal from either direction along a general interconnect line. A timing-sensitive router selects buffered and unbuffered segments as appropriate to improve the performance of the net.

Block-to-Interconnect Connections

As shown in figure 2.3.19, the XC3000 architecture has a complex interconnection scheme from the wiring channels to the CLB inputs. Rather than group all connections in a single channel as in the XC2000, the inputs are connected to segments in two wiring channels, and each input pin connects to a fraction of the

segments in the channel. A net routed in either channel can be connected to the input pin, provided it is on the proper track in the channel. Similarly, as shown in figure 2.3.20, outputs from CLBs can be connected to two different channels. The channels are at the ends of the fixed output wires shown in the figure. The output channels are not both adjacent to the CLB. This enlarges the immediate neighborhood for high speed connections between CLBs, since a signal can skip a switchbox in two of the four directions.

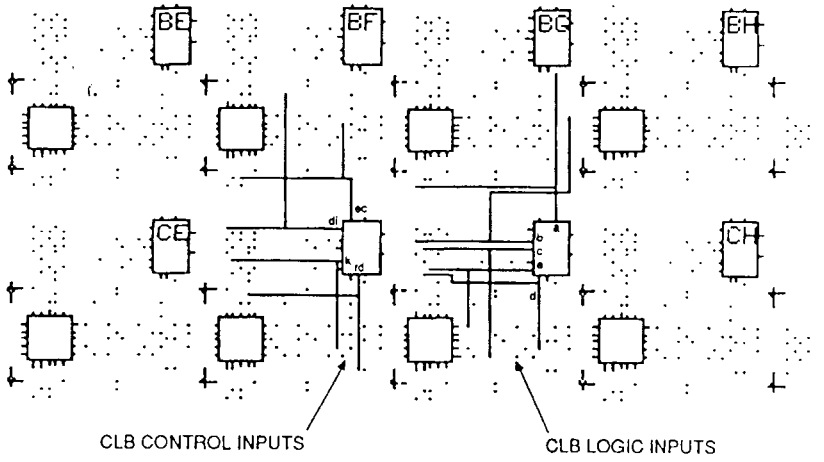


Figure 2.3.19. Input Pips in XC3000.

Both of these complexities cause placement problems. The placement software measures the quality of the placement using an estimate of routability and performance. For the XC3000, the pins are accessible from more than one channel, but there is no guarantee that the signal can be connected to the pin in any channel. Therefore, the routability depends on which channel the placer expects the router to use to route to the pin and on the ability of the router to bring the signal into the channel on the correct track.

The XC3000 Family Members

The XC3000 family consists of six members, covering more than a factor of six in capacity (see table 2). Devices are available in a variety of packages, including pin grid arrays, quad flat-packs and thin flat packs for PC-MCIA applications. Many members are available in common packages with identical footprints so designs can be migrated to higher or lower density parts without any board changes.

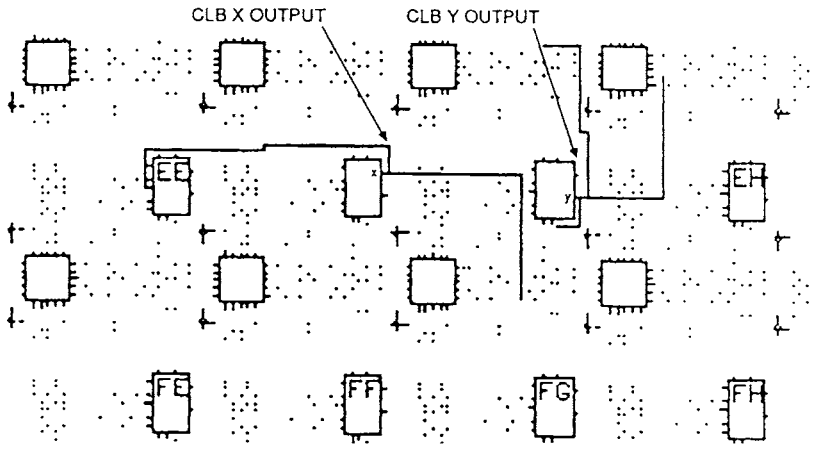


Figure 2.3.20. Output Pips in XC3000.

Table 2: XC3000 Family Members

Member	CLB Array Size	IOs	Gate Capacity	
			max	typical
XC3020	8x8	64	2 000	1 200
XC3030	10x10	80	3 000	1 800
XC3042	12x12	96	4 200	2 500
XC3064	16x14	120	6 400	3 800
XC3090	16x20	144	9 000	5 500
XC3195	22x22	168	13 000	7 500

Since its introduction in 1987, the speed of XC3000 FPGAs has increased with process migration from 1.2μm to 0.8μm feature size, and with circuit improvements to speed-critical paths. Performance of the XC3000 FPGAs over time is plotted in figure 2.3.21. Currently, the highest-speed parts are the compatible XC3100-3 (for 3ns CLB delay). This speed permits system clock frequencies around 100 MHz, although some designers have reported achieving speeds even faster than that on pipelined designs with careful partitioning, placement and routing.

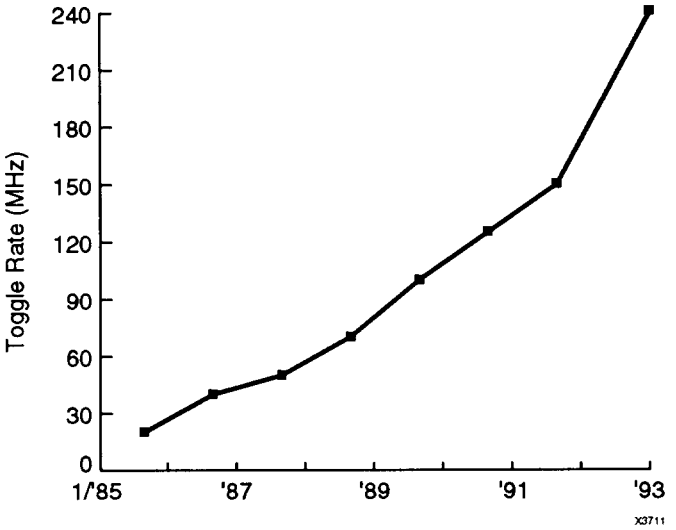


Figure 2.3.21. Historical Performance of XC3000 Parts.

Additional enhancements to the XC3000 speed and density came as a result of software improvements. These improvements do not change the maximum gate capacity, nor do they change the maximum toggle frequency, but they do increase the typical capacity and narrow the difference between the toggle frequency and the automatically-achievable system clock frequency. Software has improved the speed of automatically placed and routed designs by about 50% since the introduction of the family.

The Xilinx XC4000 Architecture

The XC4000 architecture [Hsieh 1990] [Trimberger 1991] was designed to improve performance and gate density for large designs. Several dedicated features were added to the general-purpose logic features of the XC3000, yielding an interesting combination of special-purpose and general-purpose functions. The XC4000 family was designed using placement and routing tools to evaluate architectural decisions.

Architectural features were designed to interact efficiently with an automated design methodology.

The XC4000 CLB

The XC4000 CLB (figure 2.3.22) is similar to the XC3000 CLB. It contains three lookup tables and two flip-flops. The two primary lookup tables, labelled F and G, each implement any function of four variables. These two results can be brought out of the block independently or they can be combined with another input in the H lookup table to make any function of five inputs or some functions of up to nine inputs. This allows functions such as nine-input AND, OR, exclusive-OR (parity) or address decode to be done at high speed in one block. The flip-flops can take their inputs independently from the lookup tables or from external signals, but they share control signals. Unlike the XC2000 and XC3000, flip-flop outputs are not recirculated internally. A registered feedback signal in the XC4000 must be routed in the general interconnect back to a CLB input pin.

The XC3000 can implement arithmetic with sum in one lookup table and carry in the other. The XC4000 CLB can implement arithmetic this way also, but since the speed of arithmetic operations is dominated by the speed of the carry chain, the XC4000 CLB includes dedicated high-speed carry logic. Figure 2.3.23a shows a configuration of the CLB in arithmetic mode. The F and G lookup tables compute two sums while dedicated logic calculates the carries. The dedicated carry logic in the XC4000 substantially speeds up arithmetic while doubling its density.

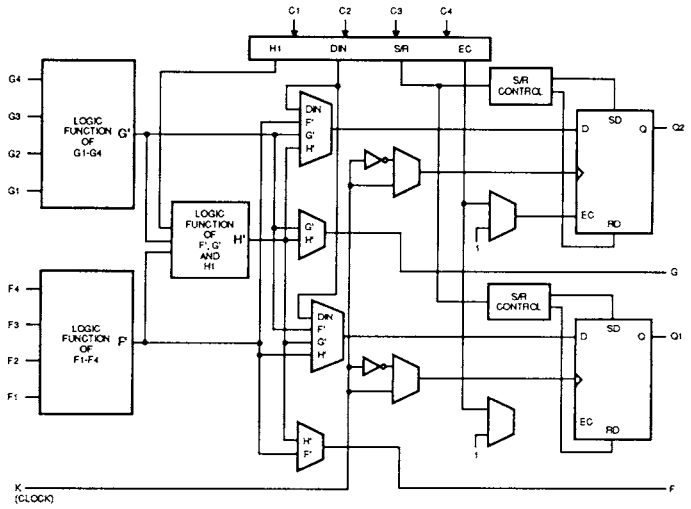


Figure 2.3.22. The XC4000 CLB.

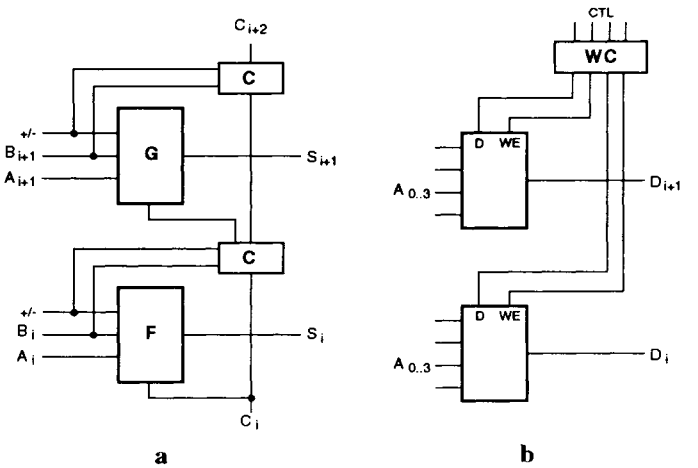


Figure 2.3.23. Special Configurations of the XC4000 CLB. a) Arithmetic. b) Memory.

Small, fast memory is an important component of many digital systems, appearing frequently as registers and FIFOs. The F and G lookup tables in the XC4000 CLB can be configured as 16x1 static memories. To write into the lookup table memory, the lookup table inputs address the memory cell to be written. Additional inputs to the CLB are write enable (WE) and data (D) (figure 2.3.23b). Reading from a memory is the same as evaluating a function. One CLB can implement 32 bits of memory, configured either as 16x2 or 32x1. On-chip memory makes FPGAs a reasonable implementation target for register-intensive digital systems. Since the memories are traded off against logic, unused memory is not wasted.

The XC4000 IO Block

Figure 2.3.24 shows the details of the XC4000 IOB. Signals to be output from the chip can be registered before output and enabled by a separate control signal. Outputs can be optionally pulled up or down, and the output driver can be configured with either fast or slow slew-rate. Inputs from the pad can be brought into the interior of the chip directly, registered or both to facilitate multiplexed bus interfaces. Furthermore, inputs can drive dedicated decoders, built into the edge interconnect, for fast recognition of addresses.

The XC4000 IOB includes boundary scan logic compatible with the ANSI IEEE 1149.1 (JTAG) boundary scan standard [IEEE 1989] [Maunder 1990]. The logic is not shown in figure 2.3.24. Boundary scan can check internal logic or external logic. Scan operations can take place before or after the FPGA is programmed and do not interfere with the operation of the part. The scan path and control signals are available

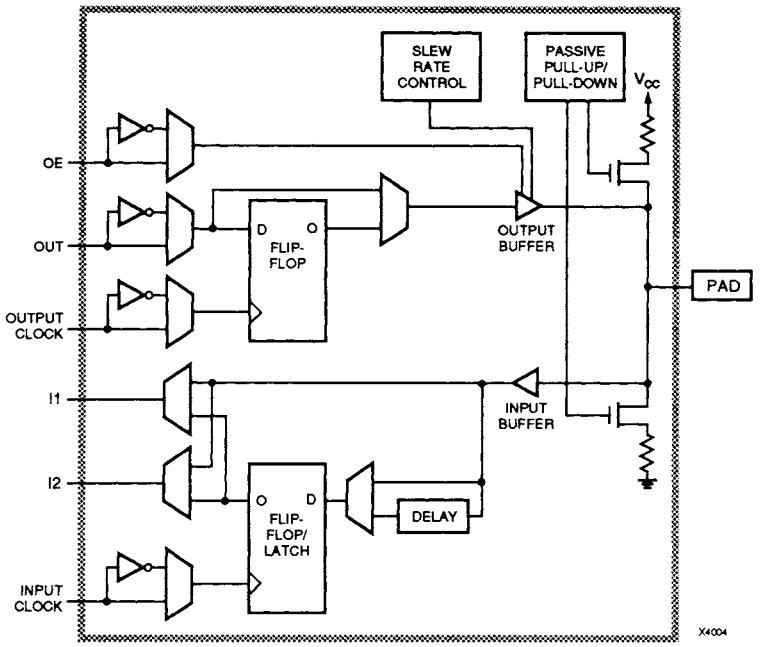


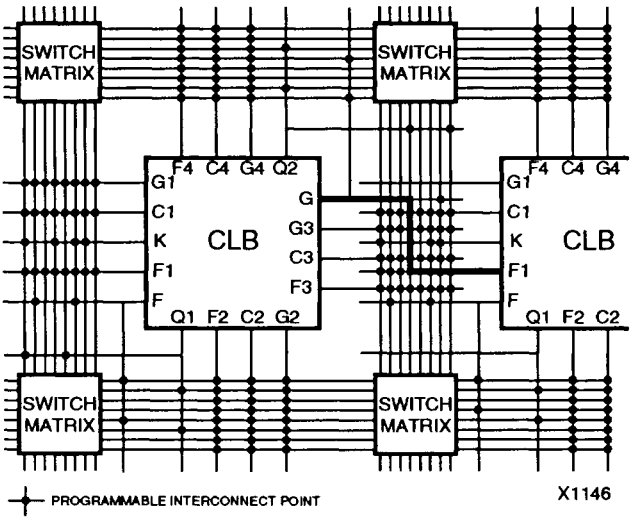
Figure 2.3.24. The XC4000 IOB.

internally in the FPGA, so additional test paths and registers can be implemented on the FPGA to make customized tests.

Wiring Architecture

Figure 2.3.25 shows an overview of the general-purpose interconnect in the XC4000. This wiring includes single-length *general-purpose* interconnect, like the XC2000 and XC3000, and *double-length lines* that bypass alternate switchboxes. Since signal delay is more dependent on the number of pips through which a signal passes than on the length of the segments, the double-length lines allow a signal to travel twice the distance in the same amount of time, or to travel a given distance in half the time.

The switchbox connections in the XC4000 are significantly fewer than those in the XC2000 and XC3000. Inside the switchbox, each segment can connect to three others, one on each of the other three sides of the switchbox (figure 2.3.26). Fewer pips means less load and faster interconnect. The drawback of fewer pips is that wiring may be more difficult. As a result, the XC4000 has more interconnect segments in the channel than an equivalent XC3000 would have.



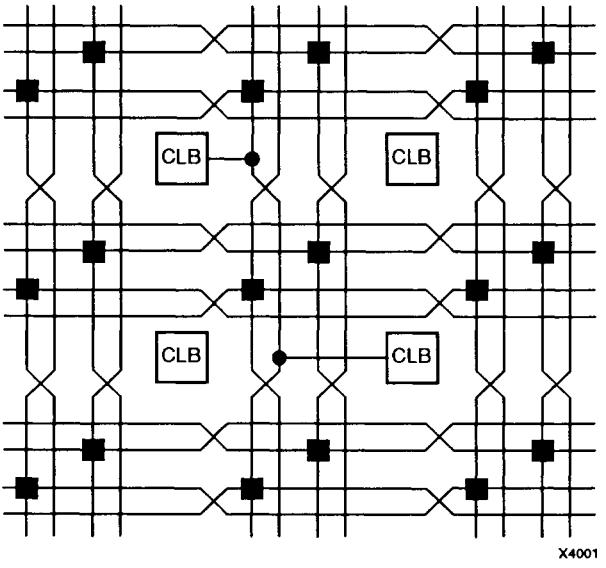


Figure 2.3.25. XC4000 Interconnect.

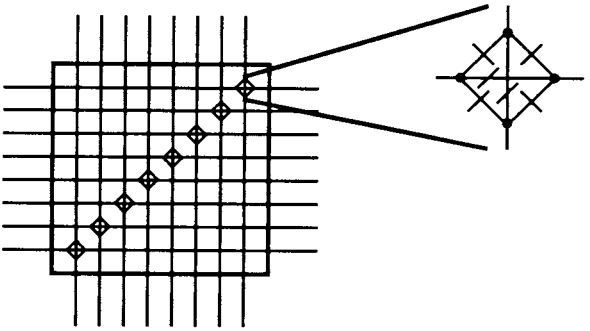


Figure 2.3.26. The XC4000 Switchbox Connections.

The XC4000 interconnect includes more long lines and global lines than were available in the XC2000 and XC3000 for high fanout and high-speed wiring. Two of the long lines in each row can be configured as three-state busses. The XC4000 long lines can be broken in the center of the chip to provide two half-long lines to improve routability. The half-long line has half the capacitance of the complete long line, thus decreasing the delay.

Four of the vertical long lines in figure 2.3.27 are the *global long lines*. Signals on global long lines can originate on-chip or off-chip. They are driven by dedicated high-

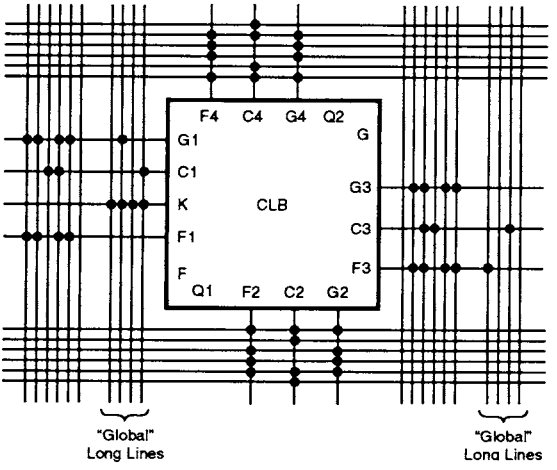


Figure 2.3.27. XC4000 Long Distance Interconnect.

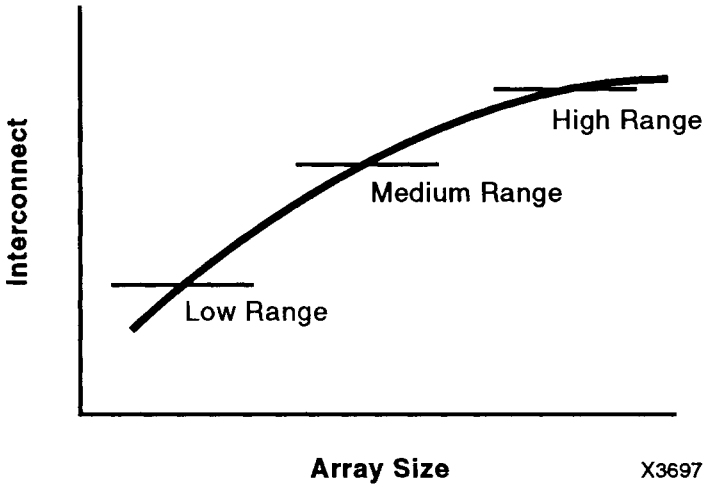


Figure 2.3.28. Interconnect Requirements Versus Array Size.

drive clock buffers and wired through the core of the chip to all CLBs to minimize clock skew.

Large arrays of blocks require proportionally more interconnect than small arrays. Therefore, the XC4000 family has three different size ranges of interconnect (see figure 2.3.28). The middle range, designed to support 14x14 to 22x22 arrays of CLBs, has eight single-length lines, four double lines and six long lines in each channel horizontally and vertically. It has four additional clock lines per column. The interconnect resources for the smaller range of parts has four single-length lines, four double lines and four long lines in each channel. The channel description for the larger parts has not been announced, but wirability estimates suggest that the differences may be significant. Using Heller's [1978] techniques, approximately 24 tracks are needed to successfully route a 25x25 array consistently.

Unlike its predecessors, the XC4000 has no redrive buffers in the interconnect. Instead, a signal requiring a repowering buffer must be routed through a CLB, either through an unused lookup table or through a repowering path that uses a CLB control input and sequential output. This design eliminates some of the complexity of the interconnect, at the cost of requiring timing-driven placement and routing software to make the trade-off between routability and performance of signals.

High-Level System Support

The array of CLBs naturally divides into two-bit slices for datapath operations (figure 2.3.29). The logic blocks contain carry logic to support two bits of addition or subtraction per block, or two 16x1 memories and two flip-flops per block. There are

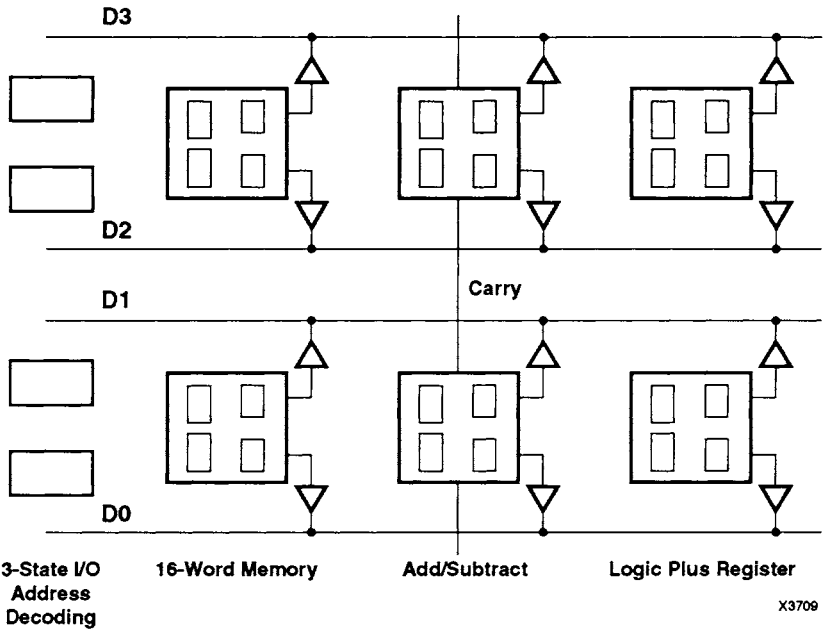


Figure 2.3.29. Bitslices in the XC4000.

two three-state bus lines per row in the array. These bus lines connect to two I/O blocks per row. I/O blocks can be configured to continue the three-state bus off-chip. Interfaces to external systems are simplified by bus-de-multiplexers, on-chip address decoders and low-skew on-chip clocking.

As a result, the XC4000 architecture conveniently supports bus-oriented datapaths, bit-sliced horizontally, two bits per row. Datapaths can contain arithmetic, memory or any other function implementable in a lookup table or a collection of lookup tables. This functionality is available in addition to the ability to build random logic, interconnected in whatever fashion is needed, using lookup tables and flip-flops.

Block-to-Interconnect Connections

The XC4000 block-to-interconnect connections are very similar to those in the XC2000. Block inputs connect to all interconnect segments in the channel, but outputs connect to only half the segments (figure 2.3.25). As in the XC3000, the CLB outputs drive signals in two different channels, but both those channels are adjacent to the block, simplifying the placement wirability estimation. The XC4000 inputs and outputs are distributed around all four sides of the CLB, eliminating the directionality bias of the XC2000 and XC3000.

The control signals to the XC4000 CLB are connected to wiring channels on all four sides. Configuration circuitry allows any of the control pins to drive any of the internal control signals, DI, CE, INIT or H1. The design software recognizes these control signals as logically equivalent, and uses pin swapping to improve delay and routability.

The XC4000 Family Members

The XC4000 family consists of ten members, covering a wide range of logic capacity (see table). Smaller parts (XC4002A through XC4005A) are available with less interconnect, and hence a smaller die. Two parts, XC4003H and XC4005H are available with increased I/O capacity.

The XC4000 CLB delay is 4.5 ns, comparable to an XC3000 toggle rate of about 180 MHz. In this architecture, systems in the 50 MHz to 60 MHz range are practical, and much faster systems are possible.

Table 3: XC4000 Family Members

Member	CLB Array Size	IOs	Gate Capacity	
			max	typical
XC4002A	8x8	64	3 000	2 000
XC4003A	10x10	80	4 500	3 000
XC4003H	10x10	160	4 500	3 000
XC4004A	12x12	96	6 000	4 000
XC4005/5A	14x14	112	7 500	5 000
XC4005H	14x14	192	7 500	5 000
XC4006	16x16	128	9 000	6 000
XC4008	18x18	144	12 000	8 000
XC4010	20x20	160	15 000	10 000
XC4013	24x24	192	20 000	13 000

Programming the FPGA

Configuration

Configuration is the process of loading the program into the FPGA. The program of an SRAM-programmable FPGA is stored in an array of memory cells that make up the lookup tables and control the pips and multiplexers. The program is loaded into the FPGA serially to minimize the number of pins required for configuration and to reduce the complexity of the interface to external memory. Internally, the configuration memory is arranged as a two-dimensional array. The program is shifted into the FPGA and assembled into a long word that is then loaded in parallel into the configuration memory. During programming, all internal drivers are disabled to avoid potential contention in the interconnect. Contention, even momentary contention, causes power surges, and the high current can damage internal metal traces.

All Xilinx FPGAs include the circuitry necessary to load the program from external memory. Essentially, every FPGA includes its programmer. The FPGAs can be configured in a number of different ways, depending on the application (figure 2.3.30). Configuration can be initiated by the chip itself when it senses power-up. In this "master" mode, the FPGA uses a simple 4-wire interface. The chip generates `clock` and `chip enable` signals to extract data from a small-footprint serial PROM which produces the program serially on the `data` line. In a system, the FPGA may be permanently mounted on the board, with the programming bitstream stored in the replaceable PROM. If the designer prefers to use an external clock to control programming, the chip can be configured in *slave* mode, where an external source generates the clock and data signals. The FPGA can emulate a simple processor peripheral for configuration, accepting an external clock, peripheral select and byte-wide data. In all modes, many chips may be connected in a daisy-chain fashion, with the first chip in the chain generating control signals and passing data to following chips. All configuration flows through the first chip and all chips become active simultaneously when programming is complete.

Besides configuring on power-up, SRAM FPGAs can be re-configured on command while residing in the circuit. This feature allows a designer to create a system in which the FPGA's program may change during operation.

Readback

Another powerful feature of the FPGA is *readback*, the ability to read out of the chip the program and also the contents of internal flip-flops, latches and memories. A working part can be stopped and its state recovered. In systems, readback is used for design verification and debugging. During manufacturing test, configuration and readback allow direct test access to observe and control internal nodes, greatly simplifying test program generation.

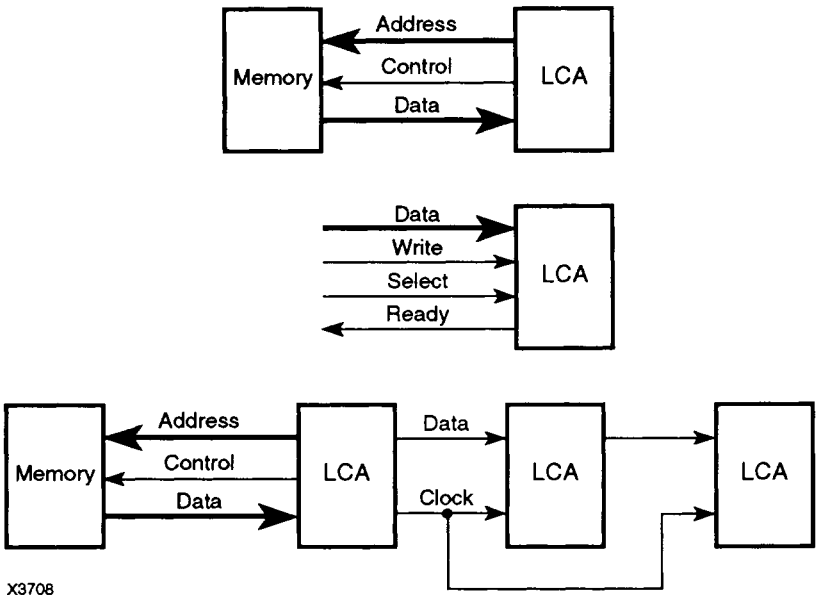


Figure 2.3.30. Configuration Patterns.

Security

The programming bitstream includes a security bit to prevent readback, so a design can be delivered without the ability to unload it from the part. In an application where secrecy is paramount, a user may configure the SRAM FPGA before shipping, and keep the part powered with a battery. The user may then remove the programming source and ship the system in which the FPGA resides. In low-power standby mode, a battery can hold the FPGA programming for years. This capability allows users the ultimate in security because there is no way to reverse-engineer the logic on the part. Programming makes no physical modification to the chip, so it cannot be recovered. If the chip is removed from the battery or power supply, the programming is erased.

2.4. Software

The design process for SRAM-based FPGAs is similar to that for MPGAs. The design process has three parts: design entry, implementation and verification (figure 2.4.1). Design entry tools used for MPGA design can also be used for FPGA design. These tools include schematic editors, state-machine generators, and synthesis tools. The interface to the FPGA design implementation system is a netlist plus implementation constraints, such as timing requirements and pad placement. Implementation tools partition the logic to fit into the FPGA, find a good placement

for the logic and route signals between the logic blocks. The implementation system generates back-annotated netlists with routing delays for timing analysis and other verification steps. Like MPGAs, FPGA designs can be verified by simulation, and reprogrammable FPGAs make in-circuit verification an attractive alternative to extensive simulation.

Automated Design Implementation

Logic implementation on the Xilinx-style FPGAs consists of three steps, partitioning, placement and routing. This section describes the three steps and the techniques that have been used to address them.

Partitioning

The *partitioning* task, also called FPGA *mapping*, is the mapping of the logic represented by the incoming netlist into the physical primitives implemented on the chip. Typical MPGA design implementation flow does not include partitioning. Instead, MPGA vendors provide a *library* of gates for schematic entry and the corresponding hard-wired cells for the physical implementation of those gates. The incoming netlist must contain only cells from that library. The cells are mapped one-

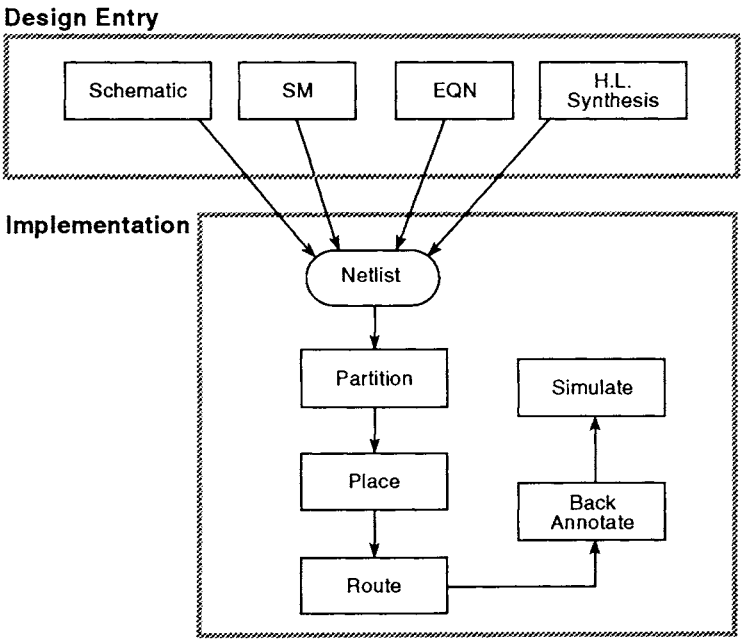


Figure 2.4.1. FPGA Design Flow.

to-one onto structures on the MPGA. While entering the logic design, the designer explicitly maps logic into the MPGA logic resources.

FPGA blocks are designed for logic density, so efficient manual mapping may be difficult. Each 4-input lookup table can represent 2^{16} functions. Each XC4000 CLB can represent approximately 10^{15} different functional patterns. Obviously, it is not reasonable to enumerate this library. Instead, the manufacturer supplies a smaller library of functions, called the "design library," and provides partitioning software to automatically map the logic in the design library into the structures on the FPGA that implement the logic. The Xilinx implementation system accepts a design in the form of a netlist of MPGA-like gates, and partitions the netlist, grouping gates into legal CLBs and IOBs for placement.

Many attempts to address this problem begin with technology mapping from logic optimization. Technology mapping is the translation of "technology-independent" logic, perhaps in the form of NAND gates, into an efficient set of gates in a target library. Several authors have reported work on mapping logic into lookup tables. Murgai [1990] uses Roth-Karp decomposition and kernel extraction in MIS-II to bound the number of inputs to nodes, then uses binate covering to group nodes into CLBs efficiently. Subsequent work [Murgai 1991] added additional covering techniques and cofactoring. Addressing performance of the resulting circuits, Murgai [1991b] decomposes the network and uses routing estimation and critical-path packing to reduce delay. Francis [1990, 1991a] decomposes the netlist into fanout-free trees that are then mapped using dynamic covering and bin packing. Additional improvements include checks for reconvergent fanout. Further work [Francis 1991b] optimized for performance by minimizing the depth of trees rather than number of lookup tables. Karplus [1991] transforms a netlist into an if-then-else directed acyclic graph (DAG), a generalization of a binary decision diagram. The DAG is then mapped using a simple marking process to achieve good results. Woo [1991] partitions a design into pieces of manageable size, then exhaustively enumerates cases to find which nets should be subsumed into lookup tables. Sawkar [1992] casts the problem as clique partitioning of the netlist. Cong [1991] [1992] decomposes the network into two-input gates and assigns a topological level (depth) starting at the primary inputs, increasing the level when the number of inputs exceeds the size of the lookup table. A backward mapping then constructs lookup tables, potentially duplicating logic to minimize network depth.

All these techniques address mapping logic into lookup tables, but do not handle the larger problem of mapping logic and flip-flops into CLBs. This problem is interesting because there is both a logical and physical component to the partitioning. The connections within a CLB are somewhat constrained, but the quality of the resulting partitioning depends on the quality of the subsequent placement, so physically-related logic should be partitioned together.

Xilinx uses two partitioning algorithms, targeted to different families of parts. The different algorithms result from the different types of intra-block constraints in the FPGA architectures.

Xnfmap is a partitioner targeted to the Xilinx XC2000 and XC3000 families. Xnfmap is driven primarily by logical constraints -- finding groups of logic that fit into a single CLB. Secondarily, Xnfmap attempts to maximize the number of nets collapsed into lookup tables and CLBs. Since collapsed nets require no interconnect, the more there are, the easier the task of routing the remainder.

Xnfmap distinguishes between combinational logic gates, which will be mapped into lookup tables, and all other logic elements, which are implemented directly by resources on the FPGA. These other logic elements include flip-flops, I/O pads and three-state buffers. Xnfmap builds groups of combinational logic by tracing back from the nets on the inputs of non-combinational-logic gates. Xnfmap adds the gate that sources the net, and recursively groups the inputs of the gate it found. The recursion stops if the net is not sourced by a combinational logic gate or if it fans-out to multiple destinations and therefore must be routed in the interconnect between CLBs. This is the case for the output of gate A in figure 2.4.2. In this way, Xnfmap groups all combinational logic into "cones" of logic that can be implemented in a lookup table.

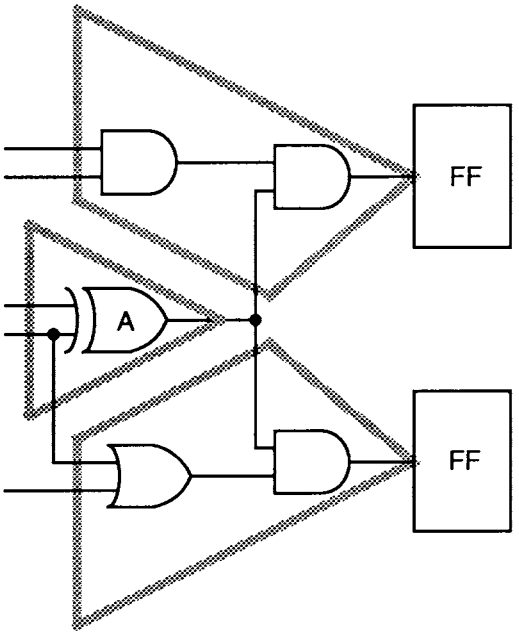


Figure 2.4.2. Cones of Combinational Logic Feeding Flip Flops.

After constructing logic groups, Xnfmap breaks up groups that are too big to fit into a single lookup table. It examines every possible division of the combinational logic to find the one that collapses the most nets, minimizes the number of groups or maximizes the use of CLB internal feedback paths. A single group may be broken several times before all of its pieces are small enough to fit into CLBs. At this point, each flip-flop is bound to the group of combinational logic that drives it.

Xnfmap then examines every possible legal pairing of flip-flops and logic blocks to find those that can be merged into a single CLB. Legal pairings are determined by the CLB architecture and are constrained by the number of outputs, the number of inputs, the number of flip-flops and the pattern of connections among them. Merging of legal pairs is *greedy*, taking those pairs that share the most inputs, thereby minimizing the number of connections for the router. Ties are broken by logical locality.

The XC4000 CLB has weaker coupling among CLB elements than the XC2000 and XC3000 architectures. Each XC4000 lookup table can be wired independently, and the outputs of the combinational and sequential elements can be routed out separately. This reduction in constraints means the XC4000 partitioner, *Blkmake*, can partition based primarily upon routability criteria rather upon than the architecture of the CLB [Trimberger and Chene 1992].

Blkmake has four steps: mapping combinational logic into lookup tables, grouping pieces of logic with pattern-matching, generating a placement-driven preferred partitioning of lookup tables and other logic, and completing the partitioning based on interconnect constraints inside the CLB. This combination of physical and logical partitioning makes a high-quality, legal partitioning.

Mapping combinational logic into lookup tables is done with a modification of Francis's Chortle technology mapper [Francis 1990]. Chortle breaks combinational logic into fanout-free trees. Since a lookup table has only a single output, the output of a fanout-free tree must be a lookup-table output. The Chortle mapper scans from logic inputs to the output using a dynamic program with variables of number of lookup tables and number of lookup table inputs used. Each step in the dynamic program finds the minimum number of lookup tables required to map the logic gate using different numbers of inputs to the lookup table that generates the gate's output. The chosen result is the one that requires the fewest number of lookup tables.

Blkmake's enhanced version of the Chortle algorithm uses support-based mapping for greater compression of reconvergent-fanout logic. It identifies lookup tables that meet the constraints for the H lookup table and preferentially accepts them during the dynamic program, increasing the number of lookup tables, but decreasing the number of CLBs required to implement those lookup tables and decreasing the delay of combinational logic.

Finally, Blkmake selectively duplicates logic to eliminate fanout nodes, when the duplicated logic improves density and performance. Gate A in figure 2.4.3a may be

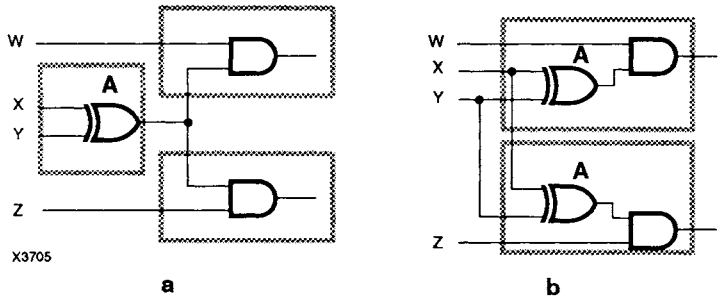


Figure 2.4.3. Duplicated Logic to Improve Performance and Density. a) Original Logic with Fanout. b) Duplicated Logic.

duplicated as shown in figure 2.4.3b, since the logic can then be implemented in two lookup tables rather than three. This logic duplication also decreases the path delay from two lookup tables with a connection between them to one lookup table with no interconnect.

Following lookup-table-mapping is a pattern-matching step to group logic that matches particularly-constrained structures in the CLB. Figure 2.4.4 shows a few of the groups that are built by pattern matching. Each group of logic is treated as a unit in subsequent placement-based optimization. The final step of pattern-matching groups lookup tables that share inputs. The grouping is greedy, grouping lookup tables with maximal sharing first.

At this point, the netlist consists of *block elements*: lookup tables, flip-flops, three-state buffers, I/O pads, clock buffers and other logical elements that correspond to physical resources on the FPGA. Some of these block elements may be grouped. The third step of partitioning uses mincut [Breuer 1977] [Fiduccia 1982] to generate a preferred placement for the block elements. Blkmake's mincut uses simple terminal propagation [Dunlop 1985] [Hartoog 1986] and a derivative of Sechen's crossing-

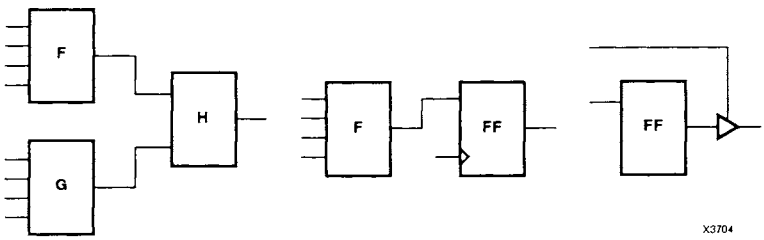


Figure 2.4.4. Pattern Matching Groups.

based cost function [Sechen 1988]. Mincut preserves the groups built by pattern-matching.

An important addition to the standard mincut algorithm is the preservation of the horizontal alignment of three-state buffer nets. The three-state buffers in the Xilinx XC3000 and XC4000 are aligned horizontally on the chip and drive a common horizontal line. Each CLB has two associated three-state buffers on two different nets. To ensure that the resulting CLBs contain matchable pairs of three-state buffers, Blkmake groups all three-state buffers that drive a common net, and treats the group as a single wide block. When the three-state buffer block no longer fits into a partition, it is locked into place. Blkmake makes a single, wide partition for the three-state bus line that contains only the three-state bus drivers and any logic that has been grouped with them. It continues to divide these new partitions, but only with vertical cut lines. Blkmake continues to partition the non-three-state logic with both vertical and horizontal cut lines.

Blkmake's mincut partitions the logic with some knowledge of the underlying FPGA architecture. It may divide the logic into uneven partitions, but maintains and manages constraints on the resources required by each of the partitions and moves the dividing line between partitions accordingly. Blkmake's mincut terminates when every partition represents a single CLB at a specific location on the target chip. However, it may not be possible to combine all the elements in the partition into a single legal CLB.

The final partitioning step is the construction of a legal CLB for the block elements in each partition. The construction is a multiple-pass greedy method that pushes overflows into adjoining CLBs. The construction is run many times from different random starting points, keeping the best result. The constructed CLB is placed at the location at which mincut placed the original partition.

Placement

The placement step accepts the partitioned design in terms of CLBs and IOBs and determines a good placement for the blocks in the FPGA array. Many MPGA placement algorithms are applicable to FPGAs with minor modifications. Most MPGA placers optimize with respect to a cost function that is based on the total expected length of all nets in the design. Net length is a reasonable measure because a shorter net length implies less interconnect area for interconnect and less capacitive load on nets. The result is that total net length correlates well with both routability and performance. However, in an architecture where the wiring can come in a few fixed lengths, as it does on all three Xilinx architectures, net length does not necessarily correlate well with either routability or performance.

Placement software recognizes two classes of constraints on placement due to the FPGA architecture: *legality* constraints which must be met to build proper functionality, and *quality* constraints that are preferred to take advantage of FPGA

features. Legality constraints are enforced in the move set. Quality constraints are handled in the cost function.

Legality constraints in the XC3000 and XC4000 dictate that three-state buffers driving the same bus must be horizontally aligned. If they are not, the three-state buffer outputs cannot drive the same net, so the placement is unroutable. In contrast, a quality constraint states that high-fanout and low-skew nets should be routed on long lines. The buffered long line delivers a signal with lower resource cost and lower delay than a solution with shorter total wire length that uses general-purpose interconnect. The placer attempts to align instances on these nets vertically or horizontally. If the instances cannot be aligned, the design is still routable, so the placer does not rigidly enforce this constraint.

Simulated annealing [Kirkpatrick 1983] has been used in many MPGA placement systems ([Sechen 1985 and 1988] and others) and is applicable to FPGA placement as well. Although simulated annealing is slow compared to constructive methods, it produces a good result regardless of the initial placement, and produces a good solution with complex, varying and possibly-conflicting constraints. The simulated annealing cost function for FPGAs contains two components for different kinds of constraints, wire length and alignment. The wire length component reduces the overall routing resource requirement, alignment allows efficient use of long lines. During placement, high-fanout nets are scored using a request/grant mechanism for long lines. The annealing cost function bases the cost for a net on whether or not it expects the net to be routed on a long line. As placement progresses, some nets may become misaligned and others more aligned, changing the placer's routing expectation which changes the way the placer calculates the cost of the nets. When placement finishes, the placer passes its routing expectation to the router as a guide to ensure agreement between the two pieces of software.

Placement for the XC2000 and XC3000 families addresses the asymmetry of the blocks by measuring wire length from the channels into which the pins connect, rather than from the blocks themselves. For those pins that connect into multiple channels, a single representative channel is used as the basis of wire length and alignment estimation.

As described earlier, Blkmake generates an initial placement for partitioned logic for the XC4000. This initial placement is improved using a Generalized Force-Directed Relaxation (GFDR) algorithm [Goto 1981]. As with simulated annealing, the GFDR cost function prefers aligned nets, boosting the effectiveness of long and double-length lines. Since moves are rejected much less often, GFDR is much faster than simulated annealing. Therefore it is run exhaustively, iterating until no improvement is found. The placement results are of comparable quality to simulated annealing, but are achieved with much less execution time.

Routing

Traditional MPGA global routing and channel routing algorithms do not work well with the FPGA architectures described in this chapter. Global routers assume that the routing problem can be decomposed spatially so that after global routing, the detailed routing of each channel can be done without interfering with other channels. This model does not work with FPGAs because the limited connections in the switchbox constrain the assignment of segments in one channel based on their assignment in other channels. For example, in figure 2.4.5, a snapshot of the XC4000 interconnect, if a net is routed on the bottom track in the lower channel, then because of the limited connections in the switch box, it can only be routed on the left-most tracks in the right and left channels and the bottom track in the upper channel. A global router that ignores these constraints will not correctly decompose the routing problem for a detailed router, and the detailed route will fail.

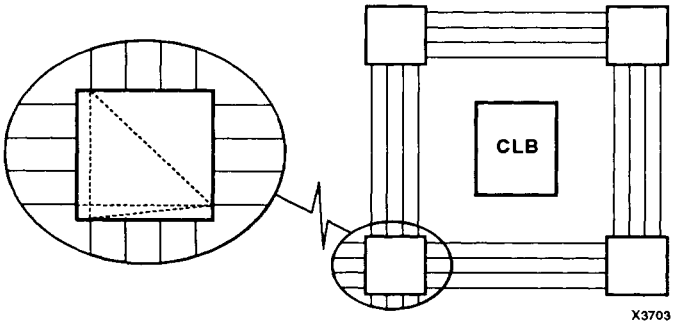


Figure 2.4.5. Routing Constraints Imposed by the Switchbox.

Channel routers assume that two non-interfering wiring layers are available for routing and that a net in a channel can change tracks when the route is infeasible due to cyclic vertical constraints [Soukup 1981]. It is possible in the architecture to provide a model of as many non-interfering wiring layers as needed. It is also possible to construct channels in which there cannot be cyclic vertical constraints. However, the limited connections in the switchbox and the limited connections from the CLBs to the interconnect channels prevent channels from being routed independently, and since the entire problem cannot be described as a single channel, channel routing is not applicable.

As a result, routing is done with an A* maze router on a graph [Nilsson 1971]. Graph nodes represent segments, and arcs represent pips and paths through CLBs. Routing proceeds from logic outputs to inputs. When searching for an input, any input to a lookup table is accepted and input pin swapping is performed. For combinational logic outputs, the router starts the maze search from both lookup tables in the CLB if

they are available, to perform output pin-swapping. This ability to swap logically-equivalent pins significantly improves the routability of designs.

The router has a detailed timing model of the FPGA, including block delays, wire segment capacitances and pip resistances. The router keeps this information in an interconnect delay model based on Penfield and Rubenstein's work [1981]. It uses an incremental method of recomputing delays during wavefront expansion to measure and to reduce delay and skew while routing. This timing-driven feature is used in the XC3000 and XC4000 to determine when to route a signal through a redrive buffer or through the repowering path in a CLB.

The FPGA router includes a rip-up and re-route pass that is used both to complete routing and to reduce delay and skew on routed nets. After routing, the implementation system may re-place and re-route the design if the routed design does not meet all timing requirements.

Path delays, not net delays, dictate the performance of circuits. Frankle [1992] described the *limit-bumping algorithm* (*LBA*), a timing-optimization method using slack allocation among source-to-load connections in the design. Required path delays are specified between flip-flops, from I/Os to flip-flops, from flip-flops to I/Os and between I/Os. Required signal times are propagated back from path destinations to net destinations. Actual signal times are propagated forward from the path sources to each net destination. The difference between required time and actual time for a connection is called the *slack* of that connection. A single connection may be on several paths, so its slack is the minimum slack calculated over all paths. These propagations can be done in $O(c)$ time, where c is the number of connections in the design.

The LBA distributes a fraction of the slack on a connection to that connection. The fraction is pre-calculated as the ratio of the connection weight to the maximum weight of all connections on any path through the connection. The weight of a connection can be simply one, or it can be a function of netlist parameters such as net fanout. As long as the calculation of the fraction is based on the netlist it need not be recalculated during routing.

The LBA uses minimum delays for each connection as an initial lower bound and calculates slacks for all paths. It iterates using the fractional distribution algorithm above to allocate the slacks on paths to allowed connection delays until all slacks are near zero. At that point, the timing-driven router routes all connections to meet their delay constraints.

If delay constraints are not met by the router, the allowable delays are increased, and slacks allocated based on the new slacks introduced by the adjustment. If timing constraints are met, and we wish to produce a faster design, timing constraints are tightened using the same iterative method. The negative slacks are allocated to connections using an "approach fraction", which is proportional to the difference

between the routed delay and the lower bound. The assumption is that a connection delay that is close to its lower bound cannot improve much, and so is not required to absorb as much of the negative slack.

The LBA has been applied to FPGA routing, with the measurement that it improves system performance by about 15% compared to connection-based timing-driven routing, and is within about 15% of optimum.

Technology-Specific Synthesis

The structured architecture of FPGAs implies preferred methods of implementing functions, particularly multi-bit functions. A schematic that is designed with the FPGA architecture in mind can be twice the density and speed as one that was designed without regard to the architecture.

Logic synthesis techniques address re-mapping random logic to a new architecture, but they do not address larger functions, such as memories and data path elements. A separate tool, called X-BLOX performs this function.

X-BLOX accepts a design as a netlist which includes variable-width multi-bit primitives. Users can specify data types: the width of busses and their encoding (such as two's complement or unsigned binary). X-BLOX determines the sizes of busses between primitives and the sizes of the primitives themselves if their widths and encodings are not specified, and checks compatibility of data types. It eliminates redundant logic and generates macros of the proper size for each of the operations. The macro generation is both technology-dependent and die-dependent. Different architectures prefer different implementations of the functions and different parts within a family have different numbers of clustered resources (such as column-aligned carry logic in the XC4000). X-BLOX generates its output accordingly, making optimized macros for each family member.

X-BLOX includes several other optimization steps related to the FPGA architecture. It moves flip flops from CLBs into IOBs where possible, assigns nets to global high-speed buffers if the device has them available.

A design using X-BLOX is simpler and faster to specify than one specified with a netlist of simple gates, because the desired functionality can be expressed in higher-level symbols. This high-level specification allows X-BLOX to optimize the high-level function to the target FPGA, so efficient design does not incur a density or performance penalty. X-BLOX can be used directly in a schematic or driven from high-level synthesis through a netlist.

Manual Design

MPGA and FPGA design systems allow manual intervention in the implementation to varying degrees. Typically, a designer can constrain placement to force signals to package pins. Additionally, a system designer can label nets as time-critical or assign

nets and paths with maximum delay values. These constraints can be provided as annotation in the schematic or in a separate design constraints file.

Manual intervention in FPGA partitioning can take the form of a CLBMAP, a schematic-level constraint that forces the partitioner to accept a user-defined mapping for part of the logic. The CLBMAP is a cell in the library that a designer connects in parallel with the logic it is mapping. The CLBMAP represents no logic, but its connections are used to guide the partitioner to implement the mapped logic in a single CLB. A designer can go further, designing in terms of CLBs with their programming.

The XACT Design Editor (XDE) (figure 2.4.6) is an interactive graphical editor similar in concept to MPGA wiring editors. XDE contains both the logical and physical descriptions of the design. Users modify both descriptions simultaneously as they design the circuit. A designer can use XDE to pre-place and route CLBs or to do post-placement and routing fixup. XDE can also be used as a complete design system, allowing a designer to map the logic manually onto the device. XDE can accept a netlist as input, or a design can be created and implemented completely in XDE.

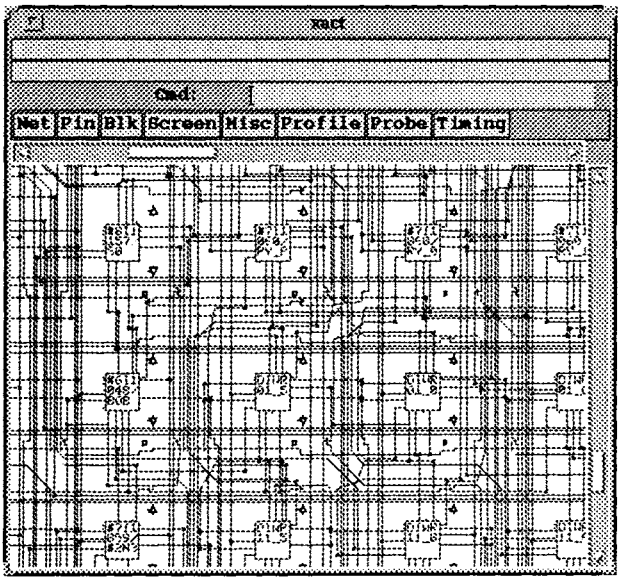


Figure 2.4.6. XDE Screen.

XDE includes the checking and editing functions required for manual design, including a design rules checker, a timing verifier and a router. XDE allows a designer to turn on or off individual pips in the interconnect, to set the functionality of function tables, and to control the functions in CLBs and IOBs.

XDE provides the front-end interface to the debugging system as well, allowing simple modifications of the design. It includes commands to generate *probe points*, internal signals routed out to unused pads to allow external access to internal nodes for debugging a design. The probe modifications are incorporated into a debugging version of the design that is then loaded into the FPGA for testing. XDE stores the probe points separately from the base design, so they can be eliminated easily when prototype debugging is complete.

2.5. The Future

FPGAs are similar to MPGAs in many respects, so there is a large body of knowledge that a researcher can draw upon to apply in this area. However, the space of FPGA architecture is large and relatively unexplored, providing many profitable areas for research. These unexplored areas may eventually yield significantly better FPGAs in terms of density and performance. Since software depends on the architecture, many software questions will opened or re-opened by FPGA architecture innovation.

Programming Technology

The CPLD-style array architectures, built with EPROM or EEPROM transistors, cannot be scaled beyond thousands of gates of logic. The array of transistors scales quadratically, as does static power consumption, while delays increase. All large-capacity EPROM devices show significantly degraded speed relative to smaller devices, as well as massive power requirements. Managing the size and power consumption requires a multi-level logic organization, such as the island-style architectures described in this chapter. EPROM transistors are only efficient when built in large arrays, so they become inefficient in these architectures. Recent attempts to extend EPROM-based architectures to large devices have separated the EPROM section into a straightforward memory array, and placed it next to an SRAM-based FPGA, basically building an SRAM FPGA with a monolithic PROM.

Large antifuse-programmed devices rely on very high reliability of the antifuses themselves. A single ten-thousand-gate antifuse-based FPGA may have over a million antifuses. Although only a few percent will actually be used by a design, the architecture relies on those few percent being correct. If they are not correct, the FPGA will fail to program correctly, and must be discarded. Discarding devices that fail to program is not a serious issue with small devices, where the parts cost ten dollars and programming yield is above 99 percent; but on large devices, the parts cost hundreds of dollars and the programming yield may be 80 percent. It is unlikely that customers will accept discarding 20% of their $500 FPGAs. The quality of antifuse manufacture limits the size of antifuse-based devices.

SRAM-programmed devices have none of these drawbacks. They scale well with technology improvements and have very low power consumption. They can be built with very high quality and fully tested at the factory. For these reasons, the capacity of

SRAM-based FPGAs will exceed the capacity of FPGAs based on other programming technologies.

Architecture

The FPGA architectures described in this chapter were developed in an attempt to balance density, performance, cost and ease-of-use goals. If one were to emphasize one or more of these goals at the expense of the others, a significantly better architecture might result. For example, wider lookup tables would allow faster functions of more variables, but increase the size of the chip, decreasing density and increasing cost. Narrower lookup tables reduce lookup table area, but would increase the number of lookup tables and associated interconnect needed to implement a function.

FPGA interconnect is comparatively expensive, both in terms of delay and area. An architecture that includes more long-distance connections would have faster interconnect, but the resulting chips might require more area for interconnect, reducing their logic capacity. Architectures with minimal interconnect resources will appear denser, but might be difficult to route. Architectures must address both integrated circuit and software goals.

The true capacity and speed of an FPGA is measured by the ability of design automation software to exploit the architecture. FPGA architectures and software must be developed simultaneously.

Software

The CAE industry has focused on the MPGA problem and has adopted a gate-like implementation model based on MPGA features. Many of the current software issues with FPGAs are a result of their non-gate-like implementation structure. This disagreement is most evident in the schematic entry library, which is a collection of gate-level primitives. The netlist generated from a schematic preserves the gate-like structure. The non-gate-like FPGA structure requires a partitioning step before the placement and routing process. Related problems in design automation have been addressed either as placement, considering only the physical constraints; or as technology mapping, considering only the logical constraints. Both sets of constraints must be solved simultaneously in order to produce implementations that are simultaneously dense and fast.

The partitioning problem is aggravated by the use of logic optimization algorithms originally designed for gate-like implementations. They often produce results that reduce speed and density rather than improve them. The reasons are varied, but traditional algorithms tend to factor logic aggressively, making more small gates; they ignore the ability of lookup tables to subsume larger amounts of logic. They also ignore routability considerations, which are of vital importance to FPGAs. New optimization algorithms are needed for lookup-table based FPGA architectures.

High-level synthesis and logic synthesis systems must target the high-level architectural features of FPGAs to gain the performance and density advantages they provide. The Library of Parameterized Macros (LPM) [Holley 1991] is an industry-sponsored standardization effort to develop an intermediate form that includes these high-level constructs. It may provide the appropriate interface between high level synthesis systems and systems-oriented FPGAs.

Placement and routing of FPGAs provides new challenges. The relatively slow FPGA interconnect structure demands true timing-driven placement and routing algorithms. Although these algorithms have been proposed for MPGA design automation, their usefulness for MPGA designs has not been great, and their adoption for FPGAs seems to be happening more quickly.

Partitioning in Space and Time

Because of the limited capacity of FPGAs, and their applicability to prototyping, FPGAs have re-kindled interest in multi-chip partitioning. There are several important problems that must be addressed, including FPGA resource estimation (logic, I/O and routing), timing and partitioning into dissimilar parts.

A farther-reaching problem is the issue of partitioning a design in time: identifying parts of a design that can be time-shared onto the FPGA, and generating separate FPGA configurations for them. At present, not only are there no algorithms, but the current design representations appear to be lacking in essential timing information. An elegant solution to this problem will allow true time-shared hardware and usher in a new era in hardware implementation.

Design Methodology

In an environment where the cost of prototyping is high, designers must rely on simulation to verify their designs. Highly-accurate simulators are very slow, while fast simulators gain speed at the expense of accuracy. An FPGA designer can replace extensive simulation with prototyping.

Reprogrammable FPGAs can be designed with a software-like iterative-implementation methodology (figure 2.5.1). The path from design to FPGA prototype is as short as a few minutes, allowing a designer to verify operation over a wide range of conditions more quickly and with more accuracy than simulation allows. Because the SRAM-based FPGA is reusable, there is no hardware cost for prototyping. The final prototype can become the production chip.

The tools and techniques required to support a prototyping-driven methodology are interesting areas of current work. Some of these tools exist already: the internal state of the SRAM-based FPGAs can be read back for verification. Design system features allow a designer to insert soft probes into the FPGA to examine internal nodes of a prototype during operation. The placement and routing algorithms can run in an

incremental mode, adding logic with minimal impact on already-routed logic. Additional features may significantly improve the utility of the incremental design methodology.

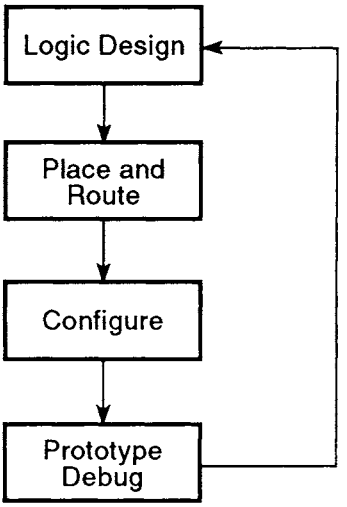


Figure 2.5.1. Incremental/Prototyping Methodology Design Flow.

2.6. Design Applications

This section begins with a discussion of general design principles that take advantage of the capabilities of SRAM FPGAs and shows how these principles are applied to canonical design examples such as state machines and counters. The discussion then covers design techniques that exploit the capabilities of FPGAs. The latter part of this section describes several designs and the techniques used to implement them. The designs demonstrate particularly effective use of SRAM FPGAs, especially with regard to the lookup tables and programmability.

General Design Issues

Design automation tools provide the designer with the ability to use an FPGA without knowing the details of the architecture. However, as with any implementation target, better understanding of the implementation and the operation of the software leads to more effective designs. This section describes some of the techniques that designers and design optimization software use to make effective use of SRAM-based FPGAs.

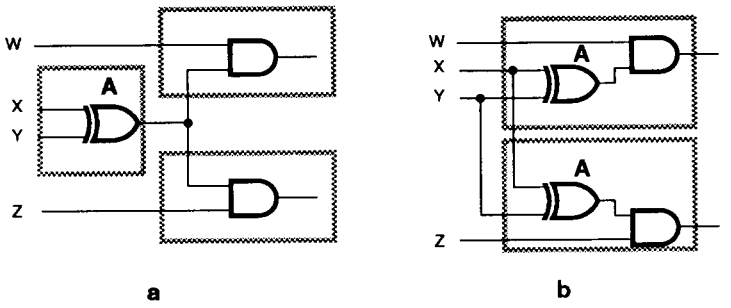


Figure 2.6.1. Logic and Partitioning (a) Before and (b) After Logic Duplication.

Duplicated Logic

Lookup tables have limited fan-in, but they implement any function of their inputs. A lookup table implements a four-input function with the same area and delay as a two-input function. A designer can take advantage of this capability by duplicating logic on the critical path. In figure 2.6.1, the shaded outlines indicate the logic that is mapped into a single lookup table. Since each lookup table has only one output, the A function must be in a separate lookup table from its destinations. In this example, duplicated logic reduces both the total number of lookup tables in the result and reduces the number of lookup tables along the path from inputs to outputs. The resulting logic is both denser and faster than the original.

One-Hot State Machines

In traditional discrete logic design, flip-flops are expensive. This is not true in the Xilinx FPGA architectures in which all combinational logic is followed by a flip-flop. The large number of flip-flops encourages one-hot encoding of state machines. In one-hot encoding, each state is represented by a separate flip-flop. A state machine with 16 states requires sixteen flip-flops, while a fully-encoded state machine requires only four. In PLD applications, where flip-flops are comparatively rare and there is no penalty for routing all state variables to all next-state calculations, this implementation would be woefully inefficient, consuming large numbers of macrocells for the flip-flops. However in an FPGA implementation, where lookup tables have limited fan-in, and where a flip-flop can follow any lookup table, the one-hot encoding often leads to a faster and smaller state machine.

One-hot encoding simplifies the next-state calculation because the states are already fully decoded. The decoded state can be gated with the transition equation for the next state in a single lookup table in one level of logic. In contrast, a complicated state encoding might require one level of logic to decode the state and a second to AND the state with the transition equation. The one-hot encoding saves one level of logic.

In addition, since the state bits connect only between states with transitions between them, and because state transitions tend to be localized, the routing of the state bits tends to be localized. The localized interconnect leads to shorter routes for the state bits, so interconnect delays are reduced.

Clock Enable

Gated clocks are known to cause problems in digital design because race conditions into the gate on the clock cannot be resolved before routing because precise delay estimates require knowledge of the routing paths of all signals. Xilinx provides clock enable signals on all flip-flops to remove the need for gated clocks. Rather than separate clocks, the design should include a single clock with the gating signal used as the clock enable on the flip-flop. This design eliminates potential glitches and allows the software to put the single clock on the dedicated global clock interconnect, which minimizes clock skew.

Performance Considerations

Since flip-flops on the FPGA are plentiful, pipelining has low cost. High-performance designs can be pipelined in single-function-generator stages that are followed by flip-flops. These techniques have resulted in designs in the XC3000 family that run at over 100 MHz, about 75% of the flip-flop toggle rate.

It is important to design at a high enough level to allow the FPGA to implement the logic effectively. The XC4000 architecture can implement a counter using the high-speed arithmetic support, but only if it is designed as a counter, not if it is designed as logic that implements the counter.

Iterative Design Methodology

Reprogrammability allows designers to experiment with the implementation technology with less effort and less expense than that required by board-level or MPGA prototypes. Designers can take advantage of reprogrammability by implementing the design one piece at a time. An effective design technique is to identify and implement performance-critical parts of a design first, lock that part in place and proceed to implement less-critical parts.

Counter Examples

This section describes different kinds of counters implemented in the XC3000/XC3100 to show a variety of density/performance points. These designs trade off counter features and density for speed. The addition or deletion of a single counter feature, such as parallel-load, can have a significant effect on the performance or density of the counter in the FPGA.

The counter designs in this section are shown at the gate level for descriptive purposes. In use, it is not necessary for an FPGA user to design counters at this level

of detail. To incorporate a preferred kind of counter in a design, he or she may select the desired counter from a library.

The most straightforward way to build a counter is as a ripple-carry adder, one bit per XC3000 CLB (figure 2.6.2a). One lookup table implements the sum and the other implements carry. The counter value is stored in one of the flip-flops in the CLB. The carry chain passes through n CLBs in an n-bit counter.

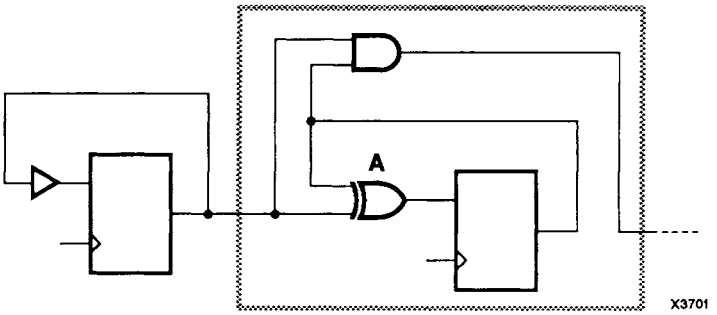
A faster ripple-carry counter can be built with the same number of CLBs, but with the carry chain passing through $n/2$ CLBs. In this counter, the basic cell consists of two CLBs, shown in figure 2.6.2b. The first CLB implements two T-type flip-flops with independent data input for pre-load. The second CLB implements two bits of the ripple carry chain. Performance can be optimized by placement that allows the carry chain to use high-speed direct connect. A 16-bit counter using this design operates at 41 MHz in an XC3100-3 device

It is possible to build a non-loadable binary counter using less than one CLB per bit by segmenting the counter into three-bit pieces (figure 2.6.3). The least-significant tri-bit piece uses two CLBs to implement the first three bits of the counter as toggle flip-flops. One lookup table in the second CLB generates a parallel clock enable signal for the subsequent bits. The second tri-bit is also implemented in a pair of CLBs. Subsequent tri-bits require three CLBs since they use the parallel clock enable signals. A 16-bit counter using this design requires only 14 CLBs and can operate at 102 MHz.

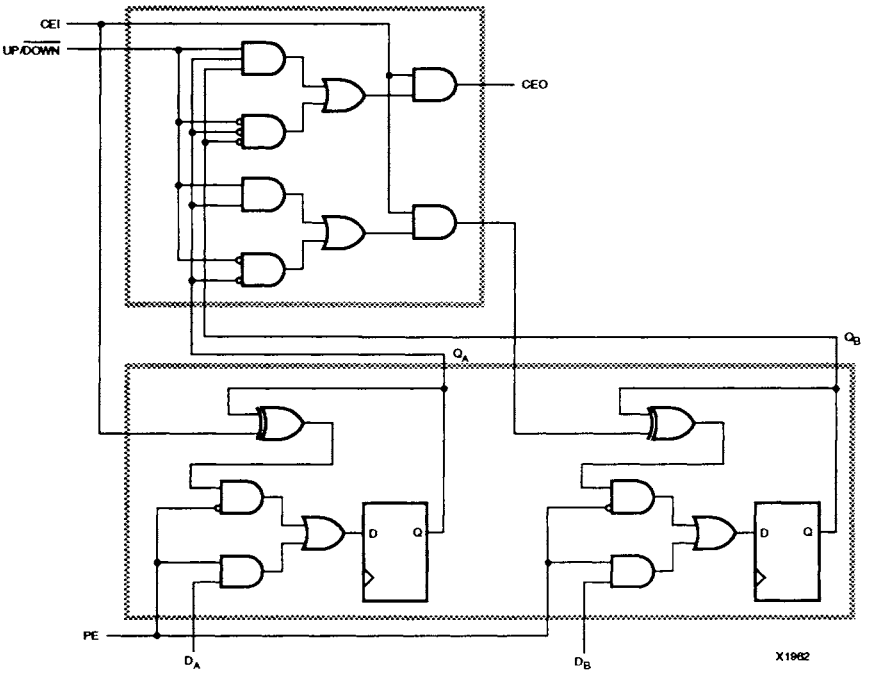
A very fast non-loadable binary counter can be built by distributing the least significant bit with a shift register (figure 2.6.4). One CLB implements a one-bit "pre-scaler," halving the effective clock rate to the rest of the counter. Unlike traditional pre-scaling techniques, the clock signal is the same to all bits of the counter. A second CLB implements a two-bit counter that generates a parallel clock enable signal, CEP2, every eight clock cycles. Ripple carry can be used for the remainder of the counter. The Q_0 pre-calculation allows two clock cycles to distribute CEP2, which is adequate if it is distributed on a long line.

In order to distribute Q_0 quickly, it is replicated once for every bit in the counter. The copies are stored in CLB flip flops that are connected in a serial chain. When the counter starts, n cycles are required to initialize the chain of Q_0 copies. The duplication of the flip flops in the CLBs allows high-speed vertical direct connections to pass the value to the next CLB in the chain and horizontal direct connections to pass the counter bits. A 16-bit counter using this design takes 24 CLBs and operates at 204 MHz.

This section dealt with binary counters. For applications that can use non-binary counters, even faster designs are possible because they eliminate carry chain delays. These counters can be implemented in devices besides the XC3000, although dedicated carry logic in the XC4000 provides a superior method of implementing



a



b

Figure 2.6.2. Ripple-Carry Counters. a) Simple Ripple-Carry Counter Built with One Bit per CLB. b) Faster Ripple-Carry Counter with One Bit per CLB.

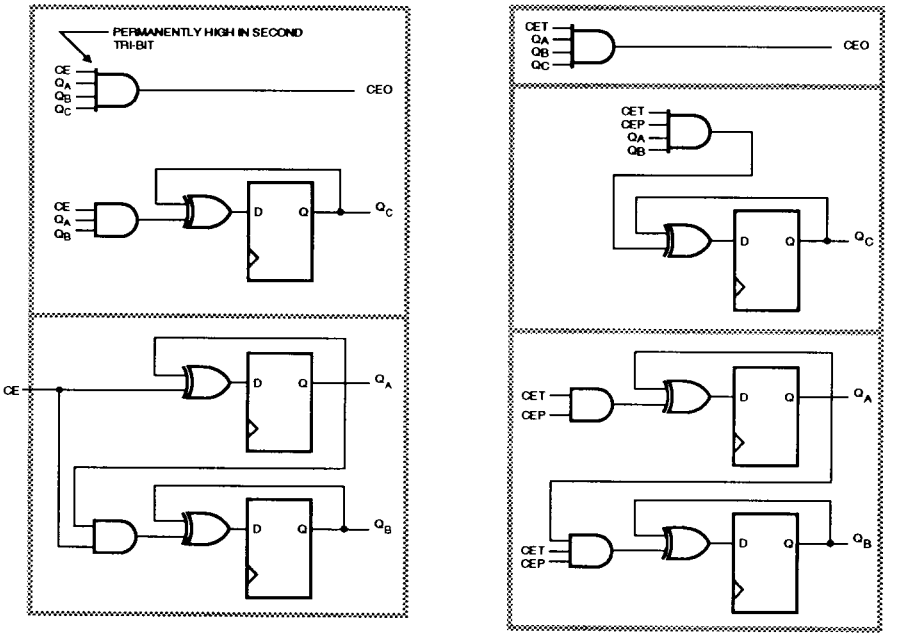


Figure 2.6.3. A Fast Binary Counter That Requires Less than One CLB Per Bit.

Table 4: Summary of 16-bit Counter Examples In an XC3100-3 Part

Type	Size (CLBs)	Speed (MHz)	Comments
Simple Ripple	16	23	loadable
Faster Ripple	17	41	loadable, up/down
Condensed	14	102	non-loadable
High-Speed	24	204	non-loadable initialization required

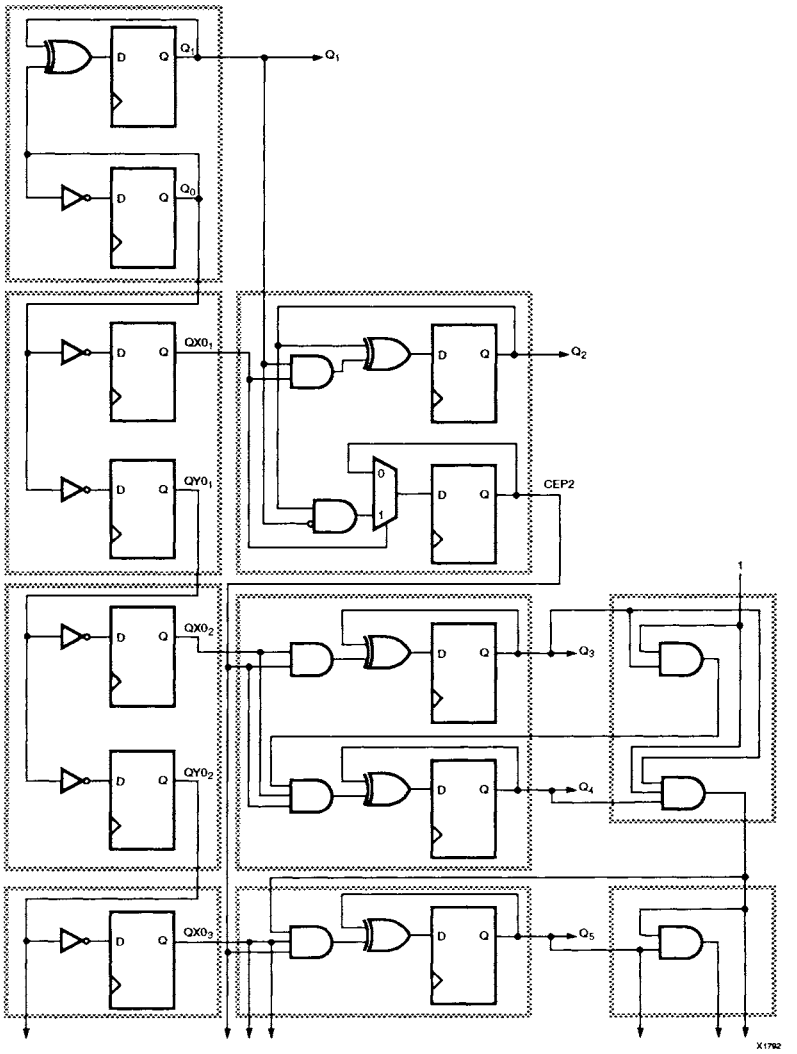


Figure 2.6.4. A Very Fast Synchronous Counter.

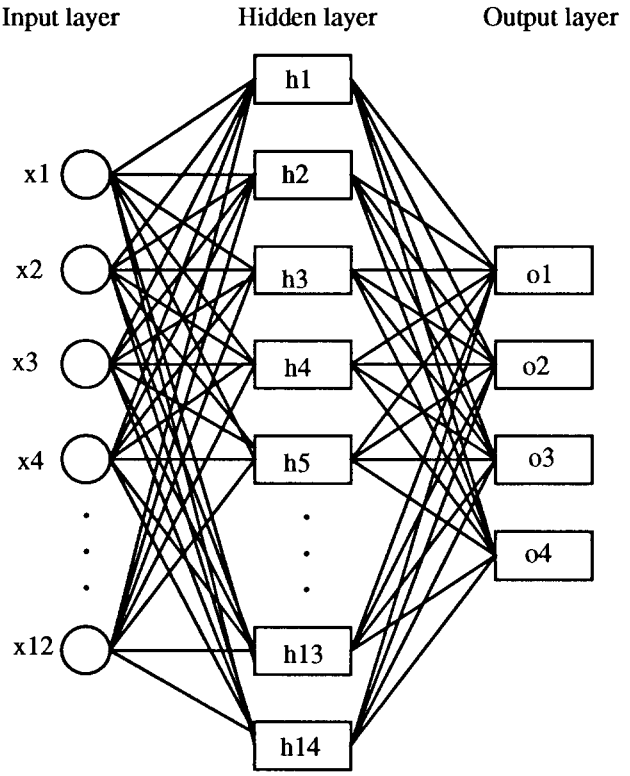


Figure 2.6.5. Ganglion Feed Forward Network (source Cox [1991]).

full-featured counters in that family. These techniques can be used to build faster counters with limited features. In the XC4000-5 speed parts, the high-speed counter runs at 111 MHz.

Efficient Multiplication by a Constant in an Artificial Neural Network

In an artificial neural network, the output of a neuron is calculated from the sum of its inputs multiplied by their respective weights. Cox [1991] used several XC3090 FPGAs to implement Ganglion, an artificial neural network for pattern matching. The Ganglion processor, shown in figure 2.6.5, implements a three-layer feed-forward artificial network. The first layer, the input layer, buffers the data from the twelve inputs. The second layer, the "hidden" layer, consists of fourteen units (neurons), each of which computes a weighted sum of the twelve inputs. The output layer computes a weighted sum of the fourteen units in the hidden layer.

Inputs are 8-bit unsigned values and the weights are 8-bit signed values. At each input to a unit, these two numbers are multiplied to produce a sixteen-bit product. The products are summed with a bias to produce a twenty-bit value which is scaled to eleven bits and passed to a large lookup table. The lookup-table implements the activation function, an arbitrary function of its input. The output of the lookup table is an eight-bit input to the next stage.

Each line connecting layers in figure 2.6.5 represents a multiplication of two eight-bit numbers. Clearly, an efficient implementation of this multiplication is crucial to the efficiency of this application. Since one of the numbers (the weight) is a constant, multiplication can be significantly simplified.

In the Ganglion design, the input eight-bit value is divided into two four-bit pieces, as shown in figure 2.6.6. Multiplication of a four-bit number by an eight bit constant produces a twelve-bit result. Each bit of the result can be expressed as a constant function of the four variable inputs bits. Therefore, each bit can be calculated in a single four-input lookup table in a single lookup-table delay. The twelve-bit result can be calculated with twelve four-input lookup tables. The two halves of the multiplication can be done in parallel, each in twelve lookup tables. The resulting partial products must then be added (with the high-order part offset by four bits). The eight overlap bits require eight full adders, the high-order four bits require four half adders to propagate the carry. The entire multiplication can be performed in thirty-two XC3000 CLBs.

In the Ganglion application, the multiplication constants may change from time to time as the network is re-trained to match new patterns. When this happens, the modification to the design is simple because only the lookup table contents change,

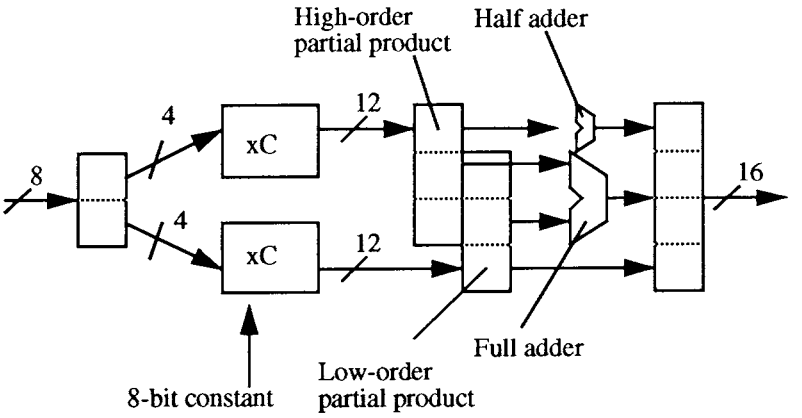


Figure 2.6.6. Multiplication by a Constant.

the interconnect does not change. The Ganglion design inserts new multiplication constants by modification of the lookup table contents in the configuration bitstream.

Distributed Arithmetic for Signal Processing

Distributed arithmetic is a method for implementing the sum-of-products that is the basis for many digital signal processing algorithms. Distributed arithmetic takes advantage of lookup tables and avoids a direct implementation of a multiplier, which would require a considerable amount of logic. This derivation is due to Mintzer [1992].

The basic sum-of-products expression for a digital filter is this:

$$Y = \sum_{k=1}^{K} A_k X_k$$

where K is the number of taps in the filter, the number of terms in the equation; and Y, A and X are multi-bit numerical values. The A_k are constants. Writing X as a sum of bits in fractional two's complement form, we get:

$$X_k = -x_{k0} + \sum_{i=1}^{B} x_{ki} 2^{-i}$$

where x_{ki} represents the ith bit of the kth number. If we substitute this expression for X into the filter function, expand the summations and re-group by powers of two, we get the following equations:

$$Y = \sum_{k=1}^{K} A_k \left(-x_{k0} + \sum_{i=1}^{B} x_{ki} 2^{-i} \right)$$

$$= -\sum_{k=1}^{K} A_k x_{k0} + \sum_{k=1}^{K} \sum_{i=1}^{B} x_{ki} A_k 2^{-i}$$

Expanding the summations and *distributing* the multiplication across the bits, we express the same function as follows:

$$
\begin{aligned}
Y = & -(x_{10}A_1 + x_{20}A_2 + \ldots + x_{k0}A_k) \\
& + (x_{11}A_1 + x_{21}A_2 + \ldots + x_{k1}A_k)\,2^{-1} \\
& + (x_{12}A_1 + x_{22}A_2 + \ldots + x_{k2}A_k)\,2^{-2} \\
& \vdots \\
& + (x_{1B}A_1 + x_{2B}A_2 + \ldots + x_{kB}A_k)\,2^{-B}
\end{aligned}
$$

Written this way, the multiplications are all single-bit multiplications where the x_{ki} bits selectively include the A_k constants. This multiplication could be implemented as iterative selective addition, but Mintzer implemented it as a table lookup. Because the A values are constants, a table of multiple-bit entries can be pre-built so the summation of each row can be done in one lookup operation.

The lookup table has 2^K entries, one entry for each of the 2^K possible sums of the A_k. The value stored in the jth location in the table is the sum of the A_k for those bits that are 1 in the bit pattern of j. Each entry in the table consists of $B\log_2 K$ bits to preserve accuracy. The table is indexed by the x_{ki} for all k for one value of i. The table lookup implements the calculation in parentheses in the equation above. If K is small (for example, 4), the lookup table can be implemented in the lookup tables in the FPGA. The lookup table for large K requires external memory.

The final step in the distributed arithmetic calculation is the summation of the partial results, each offset by one bit position. This can be done serially, re-using the lookup table for all bit calculations, as shown in figure 2.6.7. The input variables, x_k are loaded in parallel into parallel-load shift registers (PLSR). The individual bits of the variables are shifted out and used as addresses for the lookup table in sequence (for subsequent values of i). As the bits are shifted, the same lookup table calculates the partial products for successive bit positions. The lookup table results are summed serially with one-bit shift on each cycle to offset the partial products. For signed arithmetic, the partial product from the sign bit is subtracted from the total.

In a DSP application, only the first register need be a parallel-load register, the others can be loaded serially from the output of previous stage.The serial paths are shown by the broken lines in figure 2.6.7.

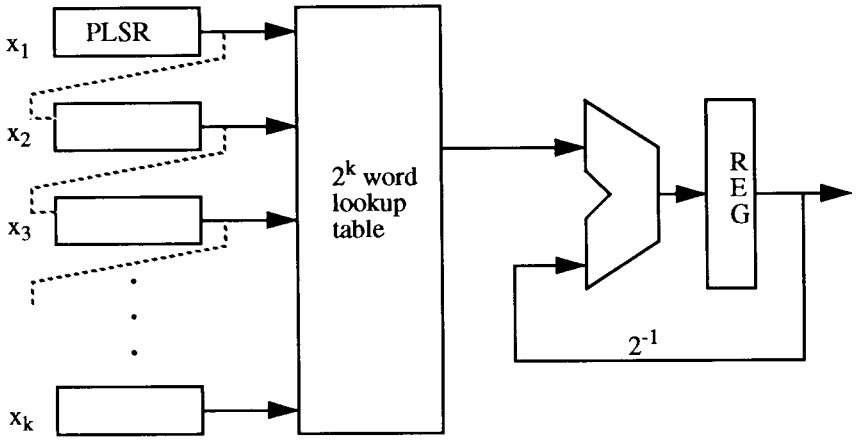


Figure 2.6.7. Distributed Arithmetic Sum-of-Products (source: Mintzer [1992]).

Mintzer built an 8-tap FIR filter using this technique. The 8-bit inputs are serialized and shifted as inputs to the lookup table. The lookup table contains only 16 6-bit words instead of 256 because of the symmetry of the filter response. The resulting filter uses 58 CLBs in an XC3042, and runs at a 1.4MHz data sample data rate.

Applications of Reprogramming

Xilinx FPGAs can be reprogrammed in 2ms to 30ms, depending on the part type. In most applications, the FPGA initiates its own programming automatically on power-up, and the application does not rely the reprogrammability of the parts. However, reprogrammability provides remarkably simple solutions to some common systems design problems.

Reprogramming for Board-Level Test

The most common system use of reprogramming is for board-level test and diagnostic circuitry. Since FPGAs are commonly used for logic integration, they naturally have connections to major subsystems and chips on the board. This puts the FPGA in an ideal location to provide system-level test access to major subsystems. The "operating" logic and the "test" logic need not operate simultaneously, so they can share the same FPGA, eliminating special-purpose test logic on the board. The system designer makes one configuration of the FPGA for normal operation and a separate configuration for test mode. The test configuration can be shipped with the board, so

the test mode can also be invoked as a diagnostic after delivery without requiring external logic.

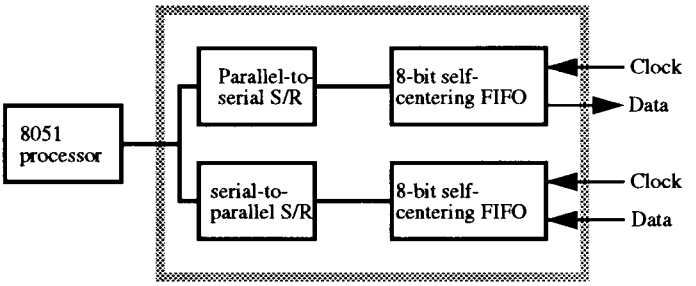
Rosendahl [1991] exploited reprogrammability for testing a bus interface built with a reprogrammable FPGA. One configuration of the FPGA is the "operating" logic, a bus interface; several additional configurations implement test circuitry. Test configurations of the FPGA include an IEEE 1149.1 boundary scan Test Access Port (TAP) for the board [Maunder 1990]. Specific tests are loaded with the TAP, including a *13N* pattern test generator for on-board RAM. The RAM tester generates addresses and patterns, captures and compares the results and keeps an error vector for later analysis.

Another test configuration of the same FPGA performs parametric tests. The test logic on the FPGA loads the external memory with a known value, then sends a read signal and address to the memory. It then samples the memory data lines after several different delays following the read strobe. The earliest match between the read data and the expected pattern gives the speed of the memory subsystem, which can be saved to become part of the system configuration. The resolution of the delay measurement can be the logic delay on chip, currently about 3ns, or it can be tuned with interconnect delays to be as fine as 1ns. Without a reprogrammable FPGA, the test and parameter measurement logic would have required thousands of additional gates of on-board logic as well as a significantly more complex board layout.

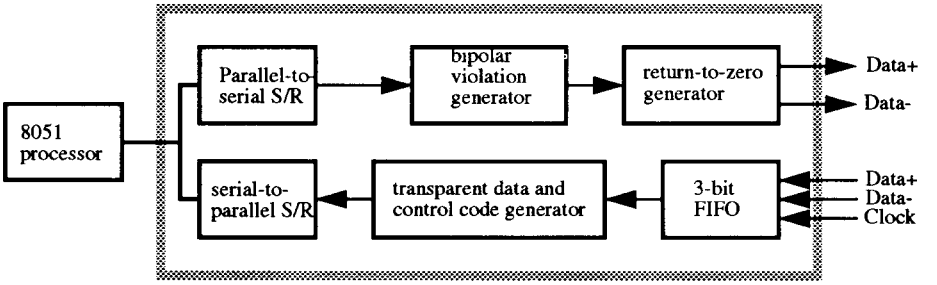
Configurable Interfaces

Test and diagnostic circuitry is a special case of a more general rule for the applicability of reprogramming: if a piece of logic can be split into multiple pieces that are not used simultaneously, then one FPGA can be time-shared to replace logic amounting to many times its maximum capacity. The system designer develops a separate FPGA program for each piece of logic, and builds the system to re-program the FPGA to switch between the pieces of logic as they are needed. This capability is especially useful in space-critical or weight-critical applications.

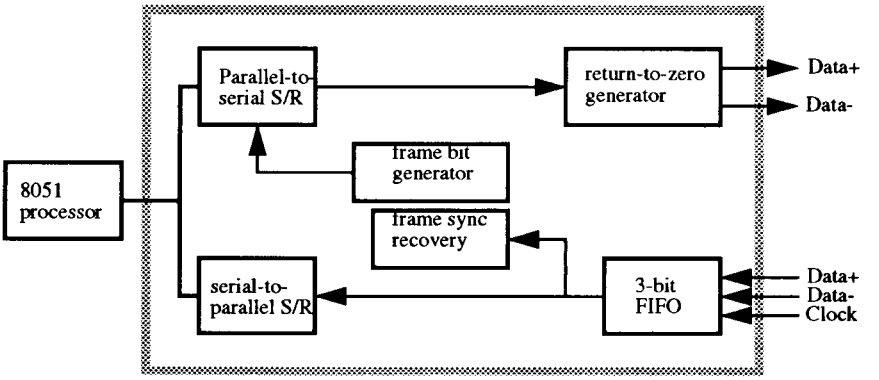
An opportunity to take advantage of reprogrammability occurs in interface design where a single system is required to interface with several different protocols. While any user of the system will use only one, the system manufacturer may wish to avoid the expense and complication of building a different piece of hardware for each protocol. The interface can be implemented in an FPGA which stores its programming in a PROM. A single PROM can contain multiple programs, and the choice of program selected during manufacturing or by the end user. This technique was used in Tellabs' channel interface card for the Crossnet 440 T1 multiplexer [Fawcett 1988]. The interface is implemented in an XC3030 FPGA, which can be configured to implement a Data Service Unit (DSU), Office Channel Unit (OCU) or secondary-mode OCU (see figure 2.6.8). Each T1 multiplexer system consists of four identical cards that can be configured independently. The end user selects the protocol



FPGA configured in DSU mode



FPGA configured in OCU mode



FPGA configured in Secondary OCU mode

Figure 2.6.8. Channel Interface Card with Different Interfaces Implemented in the FPGA.

for each card, and that selection determines the configuration that is loaded into the FPGA.

A similar technique was used by the Freeland Medical Division of Good Technologies in a video frame grabber for a PC-compatible computer. Seven FPGAs can be programmed to accept any one of the video formats used by different medical equipment. New video formats are easily added as new configurations for the FPGAs.

Hillen [1990] used an XC2018 FPGA to drive the printer interface of the Tektronix PhaserCard printer controller. The FPGA controls the flow of data from an external FIFO to the printer, formats the data, and handles signalling to the printer. Different configuration patterns for the FPGA allow the PhaserCard to control several different kinds of printers, including monochrome laser printers, ink-jet color printers, and wax-transfer color printers with parallel or serial interfaces.

When the board is idle and the printer is not printing, the FPGA is automatically reconfigured with logic to check printer status. An additional diagnostic configuration facilitates field testing.

Reconfiguration in Tape Drives

A tape drive cannot be reading and writing at the same time, so the logic for reading and writing can be implemented in the same FPGA. Liehe [1986] implemented the error correcting circuitry for a tape drive as two pieces: one for generating the error correction code when writing, one for checking the error correction codes when reading. The system reconfigures the FPGA when the tape drive switches modes. In another tape drive application [Fawcett 1993], seven FPGAs format data for different tape densities. Rather than supply different logic for each formatting style, the system programs the FPGAs to perform the proper formatting when the tape density is changed.

In this application, the FPGAs provided a second advantage: the system was originally shipped supporting only the most popular densities: 1600 and 6250 bpi. Later, the 800 and 3200 bpi formats were added and existing customers were updated in the field via floppy disk.

Reprogrammable Logic in a Configurable Display Device

The Radius Pivot monitor is a Macintosh-compatible display device that can display data either in portrait mode, preferred for word processing; or landscape mode, preferred for spreadsheets. A user of the display manually rotates the display to select the display style. Because the scan direction in the monitor is the same regardless of the orientation of the display, the display hardware must change the order of presentation of bits to the display to maintain Macintosh software compatibility.

The display interface board contains the frame buffers in video RAM, a Xilinx FPGA and the digital-to-analog converter for preparing the signal for display [Tan-Nguyen

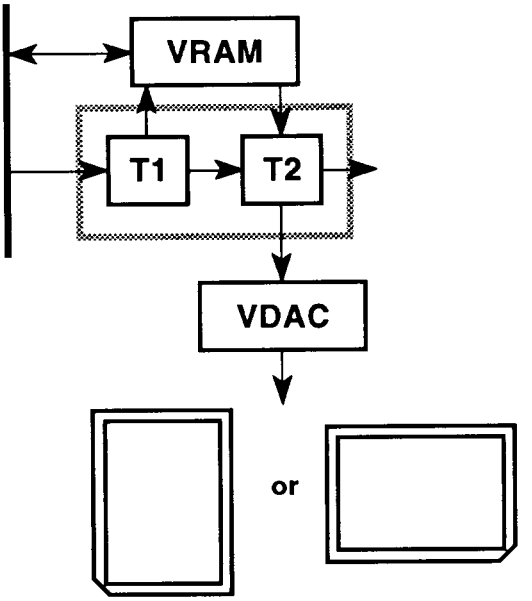


Figure 2.6.9. Pivot Display Structure.

1990] (figure 2.6.9). Part of the bitstream rotation is done when the pixels are stored in the display memory, part is done when the pixels are sent to the monitor. The data is scanned out of the video memory differently for portrait and landscape orientation. Therefore, the VRAM addresses generated by the FPGA are different in the different display modes. The final video stream goes to the monitor at 50 MHz.

The Pivot display actually has six formatting modes: one, two or four bits per pixel, with the display aligned either vertically or horizontally. These modes are all mutually exclusive, so one reprogrammable XC2018-100 FPGA with six different programs implements all six options. When the pivoting display orientation changes, an orientation-sensitive switch in the cabinet selects the correct program from a single PROM that contains all six programs and starts the reconfiguration process. Although XC2018 maximum capacity is 1800 gates, reprogrammability allows this part to replace about six thousand gates of logic.

A Fast Video Controller

Figure 2.6.10 shows a block diagram of a video controller for a full-page high-resolution display. The FPGA is required to control access to the video RAM, format the video data and generate control signals for the video monitor. Due to the high resolution, the system is required to run at a 70 MHz rate. The high throughput is

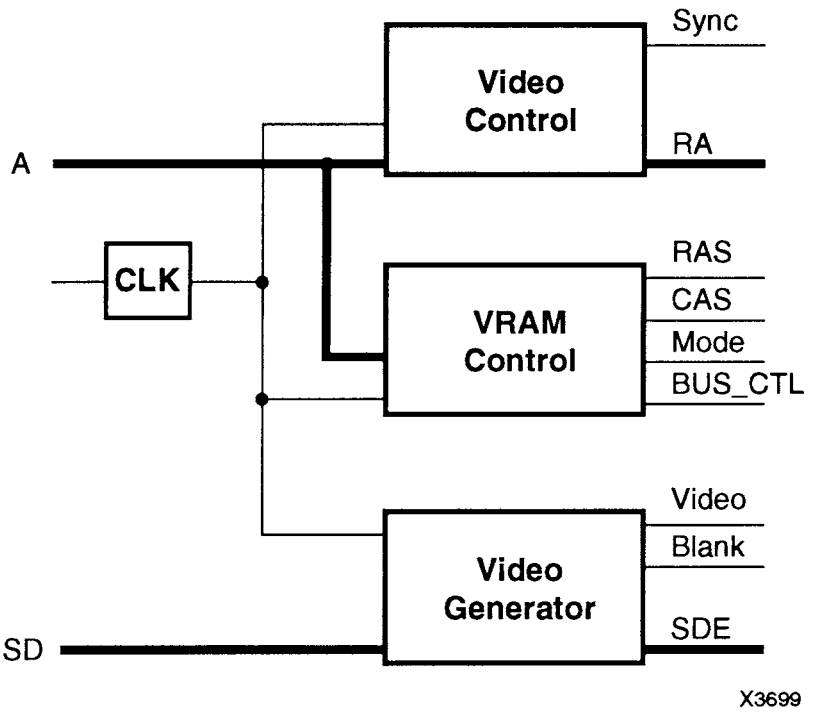


Figure 2.6.10. Seventy Megahertz Video Controller.

achieved by heavy pipelining, which is supported by the XC3000 CLB structure in which every logic block is followed by a flip-flop. To achieve the high performance, the logic was partitioned manually then placed and routed automatically. The placement and routing were done incrementally, with the few speed-critical paths placed and routed first and locked in place, before less speed critical paths were placed and routed.

This application was designed and debugged as successive prototypes in an XC3030 device, then put into production in an XC3020 device. The additional space in the XC3030 prototype device contained on-chip debugging logic, electronic "scaffolding."

A Position Tracker For a Robot Manipulator

Robot manipulator positions are tracked by counting the number of times an indicator has passed in front of a sensor. A manipulator with many degrees of freedom requires many sensors and position counters. The counters need not run fast, but they must be

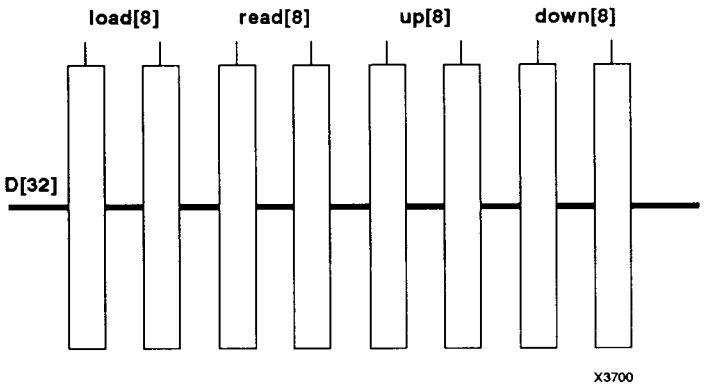


Figure 2.6.11. Robot Manipulator Position Tracker.

loadable; they must be able to count both up and down; and the system must be able to set and read the counts, the position of the manipulator, without interrupting the count.

The system in figure 2.6.11 controls a high-precision robot manipulator with 16 degrees of freedom. The range of motion of the arm and the position resolution require a 32-bit position register and counter for each degree of freedom. The design was implemented in two XC3090 FPGAs, each one implementing eight 32-bit loadable up/down counters. The position registers are in the XC3090's flip-flops, and the counters are implemented in lookup tables. Horizontal three-state lines give access to the position count registers. The position registers can be independently addressed to be read and written by the robot controller at a 30 MHz data rate. Each counter has its own up/down count signals and can count at 8 MHz. Each of the XC3090s contains over eight thousand gates of logic.

A Fast DMA Controller

Figure 2.6.12 shows a block diagram of a 16-channel DMA controller. The controller supports round-robin or priority channel selection with pipelined channel arbitration. It works with byte, word or long word transfers with separate 32-bit transfer count and address registers for each channel. The controller interfaces to 16-bit or 32-bit data busses and supports 20 million transfers per second, up to 80 MB/sec.

Figure 2.6.13 shows the block diagram of the DMA controller in an XC4008 FPGA. Registers and internal buffers use the CLB RAM configuration. Data and address sequencers use the high-speed arithmetic. Wide data busses run horizontally across the chip on three-state lines. The control logic for sequencing and DRAM interface is implemented as random logic on the same IC. The DMA controller represents approximately 7000 gates of logic and is implemented in about 200 CLBs,

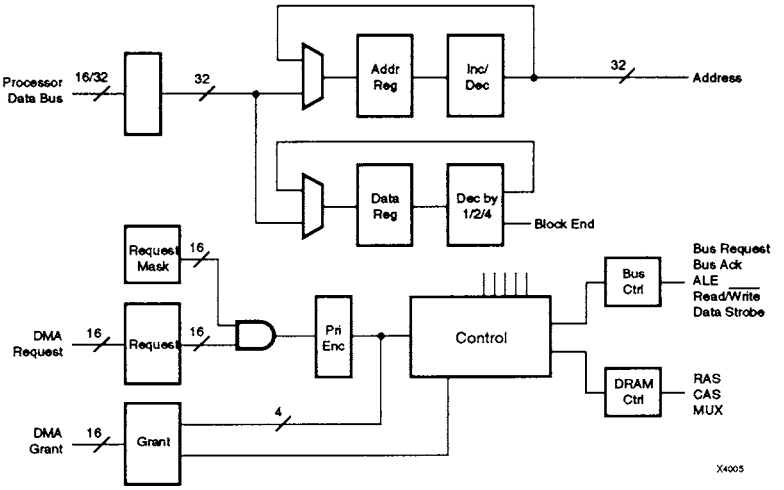


Figure 2.6.12. High-Speed DMA Controller Block Diagram.

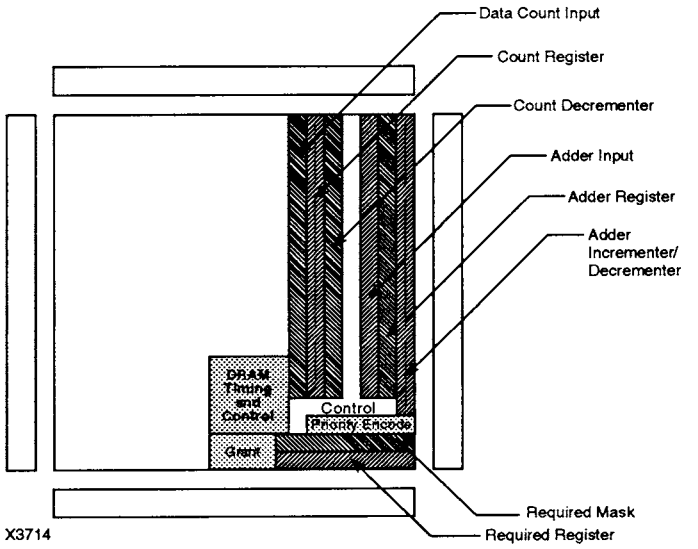


Figure 2.6.13. High-Speed DMA Controller Layout.

approximately two-thirds of the 18x18 array of CLBs in the XC4008 IC. The remaining area of the FPGA is available for other functions. A chained block scatter/gather option takes another 30 CLBs. If the DRAM is on the same board, the DRAM address multiplexer can be integrated on the XC4008 as well.

If the full functionality of this DMA controller is not needed, a simple 16-channel 32-bit DMA controller fits in only 72 XC4000 CLBs, and still runs at 20MHz.

Custom Computing Applications

SRAM-programmable FPGAs provide the opportunity to build a large-scale general-purpose system that can be reconfigured for new applications much like general-purpose microprocessors can be re-programmed with software to perform new functions. These systems have been classified as FPGAs for Custom Computing Machines, and are the subject of increasing interest as an alternative to supercomputers. This section gives an overview of a few of these systems and their innovative use of SRAM FPGAs.

The Quickturn Logic Emulator

Quickturn Design Systems, Inc. addressed the problem of prototyping large-scale ASICs and systems in the RPM Logic Emulator by using an array of Xilinx FPGAs [Walters 1990]. The RPM hardware includes a SCSI or Ethernet connection to a control processor, several emulation boards and external component adapters (figure

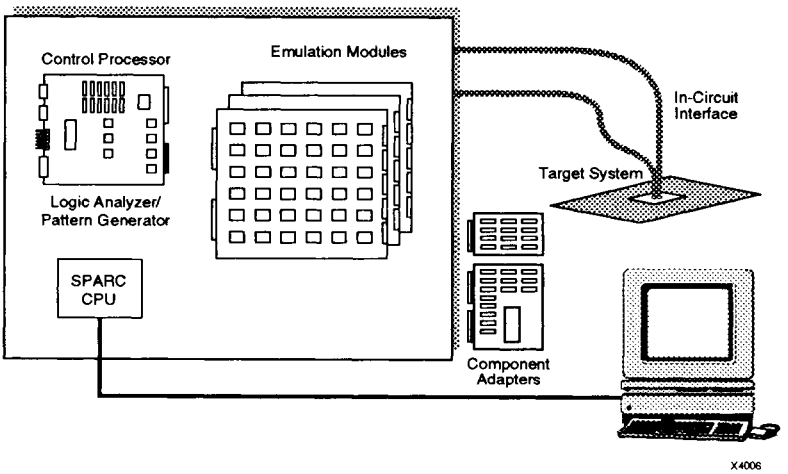


Figure 2.6.14. The RPM Emulation System.

2.6.14). Each emulation board includes an array of XC3090 FPGAs connected in a hypercube arrangement with fixed interconnects at the board level.

The external component adapters allow a designer to interface with external logic, either ICs or complete systems. The system includes configurable signal analyzers and generators, built with XC3090 FPGAs, that allow users to monitor the running system just as they would a simulation

The RPM software accepts a netlist of hundreds of thousands of gates, partitions the gates into FPGAs to optimize interconnection and density, places and routes designs on the FPGAs, and finally inserts buffers to adjust the signal timing to preserve the relative timing of the emulation with respect to the original netlist. The partitioner limits each FPGA to a small fraction of its logic capacity to ensure that enough interconnect remains in each FPGA to connect signals that pass through the design and to speed up the FPGA placement and routing that follows. Small changes are handled incrementally across all FPGAs in the system, limiting re-implementation times to minutes.

When partitioned and loaded on the emulator boards, an emulated design executes up to a million times faster than simulation.

Prototyping Hardware/Software Algorithms

Perle-0 [Shand 1989] is a platform for experimentation with logic and architectures for highly-parallel computation. The board consists of a 5x5 array of reconfigurable XC3020 FPGAs with local memory and VME bus interface circuitry to connect to a host processor. The FPGAs are connected in a two-dimensional array. Configuration for the entire array of chips is about four hundred thousand bits, with a download time of about 50ms. The array can implement functions up to about fifty thousand gates.

Perle-0 and its successor Perle-1, a 4x4 array of XC3090 FPGAs with 32 megabytes of RAM, have been programmed to perform a variety of algorithms, including image filtering, very-long-word-size arithmetic and RSA encryption. The time to implement the solutions to these problems is approximately the same as the time to implement a highly-optimized software solution to them. An algorithm is initially coded in a programming language for the host computer. The performance-critical part is re-coded into a model called a Programmable Active Memory (PAM), then the PAM model is mapped into the FPGAs. Changes in the algorithm may take weeks to describe and optimize, and design iterations can take several days. Compared to turnaround times for custom ASICs, these times are short, there are no tooling charges, and the hardware can be re-used.

Because the designers can continually improve their algorithms, Perle-0 and Perle-1 have achieved performance on several problems superior to the best custom IC solutions [Bertin 1992]. These include RSA encryption on 512-bit keys at 1500 bits per second and RSA decoding at 200 kbit/s, ten times faster than the fastest reported

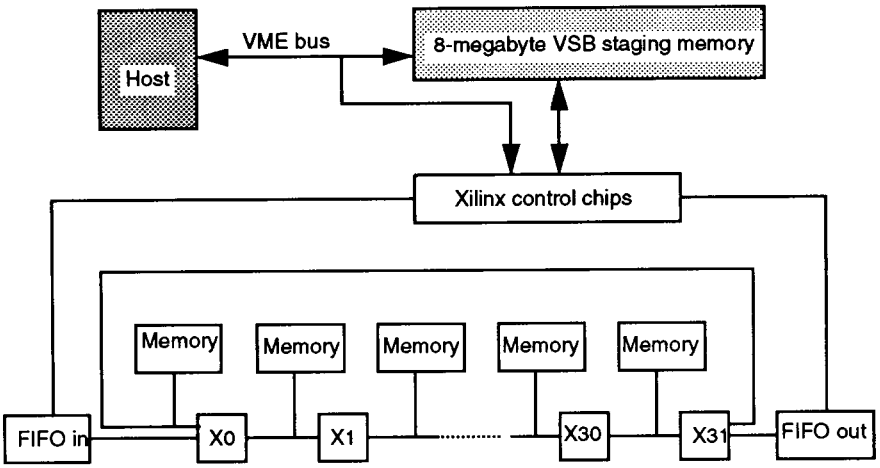


Figure 2.6.15. Splash 32-Stage Linear Array.

custom IC solution. A Perle-1 implementation of a finite difference method of computing solutions of the Heat and Laplace equations achieves performance comparable to a 25000 MIPS serial processor. A neural network emulator built on Perle-1 operates at 500 megasynapses per second.

Another example of this kind of application is the Splash processor [Gokhale 1991] [Lopresti 1991]. Splash consists of a linear array of function units (figure 2.6.15). Each function unit consists of a Xilinx XC3090 FPGA with a 128K byte external memory. The Splash processor is targeted to one-dimensional systolic problems and has been used for pattern matching DNA sequences. The general-purpose Splash board out-performs a Cray supercomputer on this problem by as much as a factor of 300 and out-performs a custom single-chip integrated circuit by a factor of 45.

Key to attaining high-quality designs in custom computing machines is the ability to repeatedly prototype the implementation, generating successively faster embedded designs, in much the same way that a software developer refines an algorithm by testing and monitoring. Since custom IC designers must eventually stop development and build the hard-wired design, they cannot continue to improve the algorithm.

A second advantage of reconfigurable systems is the ability to bring to bear large amounts of reusable hardware. Since the hardware is not dedicated to the application, is it cost-effective to bring dozens or even hundreds of FPGAs to bear on a single problem. This practice is prohibitively expensive when dealing with supercomputers, and custom ICs are typically not built to be scalable to multiple-chip solutions.

Custom computer designers build features into their systems to facilitate debugging and optimization of their highly-parallel designs. These debugging features access the

readback facilities on the FPGAs to observe internal nodes. In some cases, readback is also used to read out the results of computations, saving the need to generate extra upload logic in the FPGA array.

There have been many different methods used to program custom computers. The programming methods are related to the intended applications. The RPM engine is programmed with a gate-level netlist that is partitioned among the FPGAs in the array. The Perle-board application developers use the XACT Design Editor and other editing tools for the PAM model. The Splash board uses VHDL [IEEE 1988] as its programming language. Other programming languages have been suggested, most notably a slightly-augmented C [Thomae 1991][Van den Bout 1992].

A Flexible Processor

Wolfe and Shen [Wolfe 1988] used several reprogrammable FPGAs to implement a "flexible processor." Eight reconfigurable FPGAs are used to implement instruction decoding, address generation and datapath operations in a single-board computer. They can be configured to implement a wide variety of virtual processor architectures with different instruction sets, addressing mechanisms, pipelining schemes and ALU operations. Wolfe and Shen used the flexible processor to prototype processor architectures to solve systems of linear equations.

Custom computing machines and flexible processors may change our definition of algorithmic complexity. For example, a searching algorithm that requires $O(n^2)$ operations on a standard processor may require only $O(n)$ time on a flexible processor configured with an array of comparators. The optimal choice of processor configuration may be dynamic and problem-size dependent. The processor configuration may even change during operation.

Reprogrammability gives a system designer new capabilities: reconfigurable hardware or even virtual hardware, similar to virtual memory in computer systems. Eventually such virtual hardware may be overlaid or time-shared in a manner analogous to multiple-process-management in computer systems, employing a software-like operational methodology as well as a software-like design methodology.

In a system with tight integration between the host and the custom processor, the host processor becomes redundant. If ever one is needed, the custom computer can emulate one.

2.7. Acknowledgments

I would like to thank all those whose work and innovations are described in this chapter. You are the ones who are pushing back the frontiers of knowledge. I would also like to thank Eddie Gutierrez and Jessica Fabula for their work with drawings; Lani Sutherland for research; and Bill Carter, Peter Alfke, Bernie New, Brad Fawcett,

Philip Freidin, Mon-Ren Chene, Jon Frankle, Erich Goetting, Danesh Tavana, Lee Salutos, Steve Kelem, Kerry Pierce, Laura Smith, Samiha Mourad and John Oldfield for reviews, suggestions and assistance with technical data.

2.8. References

P. Bertin, D. Roncin, J. Vuillemin, "Programmable Active Memories: Performance Measurements," *FPGA '92. First International ACM/SIGDA Workshop on Field Programmable Gate Arrays*, 1992.

M.A. Breuer, "Min-Cut Placement," *Journal of Design Automation and Fault Tolerant Computing*, October, 1977.

B.K. Britton, D.D. Hill, W. Oswald, N.-S. Woo, S. Singh, "Optimized Reconfigurable Cell Array Architecture for High-Performance Field Programmable Gate Arrays," *IEEE 1993 Custom Integrated Circuits Conference*, 1993.

W. Carter, K. Duong, R.H. Freeman, H.C. Hsieh, J.Y. Ja, J.E. Mahoney, L.T. Ngo, S.L. Sze, "A User Programmable Reconfigurable Gate Array," *IEEE 1986 Custom Integrated Circuits Conference*, 1986.

R. Cliff, B. Ahanin, L.T. Cope, F. Heile, R. Ho, J. Huang, C. Lytle, S. Mashruwala, B. Pedersen, R. Raman, S. Reddy, V. Singhal, C.K. Sung, K. Veenstra, A. Gupta, "A Dual Granularity and Globally Interconnected Architecture for a Programmable Logic Device," *IEEE 1993 Custom Integrated Circuits Conference*, 1993.

P. Chow, S.O. Seo, D. Au, T. Choy, B. Fallah, D. Lewis, C. Li, J. Rose, "A 1.2 μm CMOS FPGA using Cascaded Logic Blocks," *Proceedings of the Oxford 1991 International Workshop on Field Programmable Logic and Applications*, W.R. Moore and W. Luk, ed., Abingdon EE&CS Books, 1991.

C.E. Cox and W.E. Blanz, "Ganglion, A Fast Hardware Implementation of a Connectionist Classifier", *Proceedings of the Custom Integrated Circuits Conference*, 1991.

M.M. Denneau, "The Yorktown Simulation Engine," *Proceedings of the 19th Design Automation Conference*, 1982.

W.E. Donath, "Placement and Average Interconnection Lengths of Computer Logic", *IEEE Transactions on Circuits and Systems*, April 1979.

A.E. Dunlop, B.W. Kernighan, "A Procedure for Placement of Standard-Cell VLSI Circuits," *IEEE Transactions on CAD*, vol CAD-4, January, 1985.

C. Ebeling, G. Borriello, S.A. Hauck, D. Song and E.A. Walkup, "TRIPTYCH: a New FPGA Architecture," *Proceedings of the Oxford 1991 International Workshop on Field Programmable Logic and Applications*, W.R. Moore and W. Luk, ed., Abingdon EE&CS Books, 1991.

A. El Gamal, "Two-Dimensional Stochastic Model for Interconnections in Master Slice Integrated Circuits," *IEEE Transactions on Circuit sand Systems*, vol. CAS-28, no. 2, February, 1981, pp 127-138.

B. Fawcett, "Taking Advantage of Reconfigurable Logic," *High Performance Systems Programmable Logic Guide*, 1989. See also [Xilinx 1992].

C.M. Fiduccia, R.M. Mattheyses, "A Linear-Time Heuristic for Improving Network Partitions," *Proceedings of the 19th Design Automation Conference*, 1982.

R.J. Francis, J. Rose, K. Chung, "Chortle: A Technology Mapping Program for Lookup Table-Based Field-Programmable Gate Arrays," *Proceedings of the 27th Design Automation Conference*, 1990.

R.J. Francis, J. Rose, Z. Vranisec, "Chortle-crf: Fast Technology Mapping for Lookup Table-Based Field-Programmable Gate Arrays," *Proceedings of the 28th Design Automation Conference*, 1991. a.

R.J. Francis, J. Rose, Z. Vranisec, "Technology Mapping of Lookup Table-Based FPGAs for Performance," *IEEE International Conference on Computer-Aided Design*, 1991. b.

J. Frankle, "Iterative and Adaptive Slack Allocation for Performance-driven Layout and FPGA Routing", *Proceedings of the 29th Design Automation Conference*, 1992.

F. Furtek, G. Stone, I. Jones, "Labyrinth: A Homogeneous Computational Medium," *IEEE 1990 Custom Integrated Circuits Conference*, 1990.

M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti, "Building and Using a Highly Parallel Programmable Logic Array," *Computer*, January 1991.

S. Goto, "An Efficient Algorithm for the Two-Dimension Placement Problem in Electrical Circuit Layout," *IEEE Transactions on Circuits and Systems*, January, 1981.

J. Greene, V. Roychowdhury, S. Kaptanoglu, A. ElGamal, "Segmented Channel Routing," *Proceedings of the 27th Design Automation Conference*, 1990.

M. Hartoog, "Analysis of Placement Procedures for VLSI Standard Cell Layout", *Proceedings of the 23rd Design Automation Conference*, 1986, pp 314-319.

N. Hastie, R. Cliff, "The Implementation of Hardware Subroutines on Field Programmable Gate Arrays," *IEEE 1990 Custom Integrated Circuits Conference*, 1990.

S. Hauck, G. Borriello, S. Burns, C. Ebeling, "Montage: An FPGA for Synchronous and Asynchronous Circuits," *2nd International Workshop on Field-Programmable Logic and Applications*, 1992.

W.R. Heller, W.F. Mikhail, W.E. Donath, "Prediction of Wiring Space Requirements for LSI," *Journal of Design Automation and Fault Tolerant Computing*, May 1978.

D. Hill and N-S. Woo, "The Benefits of Flexibility in Look-up Table FPGAs," *Proceedings of the Oxford 1991 International Workshop on Field Programmable Logic and Applications*, W.R. Moore and W. Luk, ed., Abingdon EE&CS Books, 1991.

D.D. Hill, B.K. Britton, B. Oswald, N.-S. Woo, S. Singh, T. Poon, B. Krambeck, "A New Architecture for High-Performance FPGAs", *2nd International Workshop on Field-Programmable Logic and Applications*, IFIP, 1992.

K. Hillen, B. Fawcett, "Build Reconfigurable Peripheral Controllers", *Electronic Design*, March 6, 1990.

M. Holley, C. Kaplinsky, "Streamline programmable-logic design with the proposed LPM standard," *Electronic Design*, v 39, no 21, November 7, 1991, pp 89-96.

P. Horowitz and W. Hill, *The Art of Electronics, Second Edition*, Cambridge University Press, 1989.

H.C. Hsieh, K. Duong, J.Y. Ja, R. Kanazawa, L.T. Ngo, L.G. Tinkey, W.S. Carter, R.H. Freeman, "A Second Generation User-Programmable Gate Array", *IEEE 1987 Custom Integrated Circuits Conference*, 1987.

H.C. Hsieh, K. Duong, J.Y. Ja, R. Kanazawa, L.T. Ngo, L.G. Tinkey, W.S. Carter, R.H. Freeman, "A 9000-Gate User-Programmable Gate Array", *IEEE 1988 Custom Integrated Circuits Conference*, 1988.

H.C. Hsieh, W.S. Carter, J. Ja, E. Cheung, S. Schreifels, C. Erickson, P. Freidin, L. Tinkey, R. Kanazawa, "Third Generation Architecture Boosts Speed and Density of FPGAs", *IEEE 1990 Custom Integrated Circuits Conference*, 1990.

IEEE, *IEEE Standard VHDL Language Reference Manual, IEEE Std. 1076-1987*, 1988.

IEEE Computer Society Test Technology Technical Committee, *IEEE Std. 1149.1-1990. Standard Test Access Port and Boundary-Scan Architecture*, IEEE, New York, 1990.

K. Karplus, "Xmap: A Technology Mapper for Table-lookup Field-Programmable Gate Arrays," *Proceedings of the 28th Design Automation Conference*, 1991.

K. Kawana, H. Keida, M. Sakamoto, K. Shibata, I. Moriyama, "An Efficient Logic Block Interconnect Architecture for User-Reprogrammable Gate Array," *IEEE 1990 Custom Integrated Circuits Conference*, 1990.

T. Kean, *Configurable Logic: A Dynamically Programmable Cellular Architecture and its VLSI Implementation*, Ph.D. Dissertation, University of Edinburgh, Department of Computer Science, CST-62-89, 1989.

S. Kirkpatrick, C.D. Gelatt, Jr., M.P. Vecchi, "Optimization by Simulated Annealing," *Science*, 13 May 1983.

T. Liehe, "Two, Two, Two Chips in One," *Electronic Engineering Times*, November 17, 1986.

D. Lopresti, Rapid Implementation of a Genetic Sequence Comparator Using Field-Programmable Logic Arrays, *Advanced Research in VLSI: Proceedings of the 1991 University of California Santa Cruz Conference*, The MIT Press, 1991.

K. Lyons, "A Comparison of CMOS Static Random-Access-Memory Cells," *Electronic Engineering Times*, August 5, 1985.

C.M. Maunder, R.E. Tulloss, *The Test Access Port and Boundary-Scan Architecture*, IEEE Computer Science Press, 1990.

L. Mintzer, "FIR Filters with the Xilinx FPGA", *First International ACM/SIGDA Workshop on Field Programmable Gate Arrays*, 1992.

R. Murgai, Y. Nishizaki, N. Shenoy, R.K. Brayton, A Sangiovanni-Vincentelli, "Logic Synthesis for Programmable Gate Arrays," *Proceedings of the 27th Design Automation Conference, 1990.*

R. Murgai, N. Shenoy, R.K. Brayton, A Sangiovanni-Vincentelli, "Improved Logic Synthesis Algorithms for Table Look-Up Architectures," *IEEE International Conference on Computer-Aided Design*, 1991. a.

R. Murgai, N. Shenoy, R.K. Brayton, "Performance-Directed Synthesis for TAble Look Up Programmable Gate Arrays," *IEEE International Conference on Computer-Aided Design*, 1991. b.

H. Muroga, H. Murata, Y. Saeki, T. Hibi, Y. Ohashi, "A Large Scale FPGA with 10K Core Cells with CMOS 0.8 μm 3-Layered Metal Process," *IEEE 1991 Custom Integrated Circuits Conference*, 1991.

N.J. Nilsson, *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill, 1971.

P. Penfield, Jr., J. Rubenstein, "Signal Delay in RC Tree Networks," *Proceedings of the 18th Design Automation Conference*, 1981.

J. Rose, R.J. Francis, D. Lewis, P. Chow, "Architecture of Field Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 5, October 1990.

J. Rose, S. Brown, "Flexibility of Interconnection Structures for Field-Programmable Gate Arrays," *IEEE Journal of Solid-State Circuits*, vol. 26, no. 3, March 1991, pp 277-282

G. Rosendahl, T. Paille, D. Freiling, R. McLeod, "In System Reprogrammable LCAs Provide a Versatile Interface for a DSP Based Parallel Machine," *Proceedings of the*

Oxford 1991 International Workshop on Field Programmable Logic and Applications, W.R. Moore and W. Luk, ed., Abingdon EE&CS Books, 1991.

P. Sawkar and D. Thomas, "Area and Delay Mapping for Table-Look-Up Based Field Programmable Gate Arrays," *Proceedings of the 29th Design Automation Conference*, 1992.

C. Sechen, A. Sangiovanni-Vincentelli, "The TimberWolf Placement and Routing Package," *IEEE Journal Solid State Circuits*, 20:510-522, 1985.

C. Sechen, "Chip-planning, Placement, and Global Routing for Macro/Custom Cell Integrated Circuits Using Simulated Annealing," *Proceedings of the 25th Design Automation Conference*, pages 73-80, 1988.

C. Sechen and D. Chen, "An Improved Objective Function for Mincut Circuit Partitioning," *IEEE International Conference on Computer-Aided Design*, 1988.

M. Shand, P. Bertin, J. Vuillemin, "Resource tradeoffs in fast long integer multiplication", *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, 1990.

J. Soukup, "Circuit Layout," *Proceedings of the IEEE*, October 1981.

J. Tan-Nguyen, T. Oyama, N. Moss, "Pivoting Monitor Increases Versatility of Workstations," *Computer Technology Review*, November 1990.

D. Thomae, T. Petersen and D. Van den Bout, "The Anyboard Rapid Prototyping Environment," *Advanced Research in VLSI: Proceedings of the 1991 University of California Santa Cruz Conference*, The MIT Press, 1991.

S. Trimberger, "Beyond Logic -- FPGAs for Digital Systems," *Proceedings of the Oxford 1991 International Workshop on Field Programmable Logic and Applications*, W.R. Moore and W. Luk, ed., Abingdon EE&CS Books, 1991.

S. Trimberger, "A Small, Complete Mapping Library for Lookup Table-Based FPGAs," *2nd International Workshop on Field-Programmable Logic and Applications*, 1992.

S. Trimberger, "A Reprogrammable Gate Array and Applications," *Proceedings of the IEEE*, July 1993.

S. Trimberger, M.-R. Chene, "Placement-Based Partitioning for Lookup-Table-Based FPGAs," *Proceedings of ICCD '92 International Conference on Computer Design, VLSI in Computers and Processors*, 1992.

D. Van den Bout, J. Morris, D. Thomae, S. Labrozzi, S. Wingo, and D. Hallman, "Anyboard: An FPGA-based, reconfigurable system," *IEEE Design and Test of Computers*, September 1992.

S. Walters, "Reprogrammable Hardware Emulation Automates System-Level ASIC Validation", *Wescon/90 Conference Record*, 1990.

A. Wolfe and J.P. Shen, "Flexible Processors: A promising application-specific processor design approach", Technical Report, Carnegie-Mellon University, 1988.

N.-S. Woo, "A Heuristic Method for FPGA Technology Mapping Based on Edge Visibility," *Proceedings of the 28th Design Automation Conference*, 1991.

Xilinx, *The Programmable Gate Array Data Book*, Xilinx, 1989, 1991, 1992, 1993.

Xilinx SMTO, "Static Memory Technology Overview,", Xilinx Technical Brief, Xilinx, 1991.

Chapter 3
Antifuse Programmed FPGAs

Dennis McCarty and Telle Whitney, Actel Corporation

3.1 Introduction

The architecture of an FPGA is determined, in large part, by the programmable switch technology used to configure it. Many such technologies have been considered for use in FPGAs, including laser programming [Smith] [Allen], pass transistors controlled by SRAM [Hsieh] [Carter] or EPROM cells [Wong] and antifuses [Gerzberg] [Hamdy] [Whitten].

This chapter describes the architecture, technology and use of FPGAs based on an electrically programmable two-terminal device known as an *antifuse* [Hamdy]. An antifuse device irreversibly changes from a high to a low resistance when a programming voltage is applied across its terminals. Antifuses offer several unique features for FPGAs, most notably their low *on* resistance of 100 to 600 ohms, and their small size. The layout area of an antifuse cell is generally smaller than the pitch of the metal lines it connects. It is about the same size as a via used to connect metal lines in a mask programmed array. More than 1,000,000 antifuses can now be integrated on a single FPGA, facilitating the development of routing architectures approaching the flexibility and scaling potential of conventional gate arrays. Current antifuse FPGAs offer complexity equivalent to an 10,000-gate conventional gate array and typical system clock speeds of up to 75 MHz.

While the focus here is on the Actel FPGA families similar principles will likely apply to other antifuse-based FPGAs now emerging. These developments include both new antifuse process technology and new FPGA architectures. The Act1[ElAyat] [ElGamal1] family ranges from 1200-2000 gates, the Act2 [Ahrens] family from 2500-8000 gates, and the Act3 [Schlageter] [Whitney] family ranges form 1500-10,000 gates.

Figure 3.1.1 shows a simplified block diagram of an ACT FPGA. Rows of logic modules are interspersed with horizontal routing channels containing predefined wiring segments of various lengths and offsets. Other wiring segments run vertically through the modules and across the channels. Each logic module computes a single-output function of several inputs. Each module input is connected to a dedicated

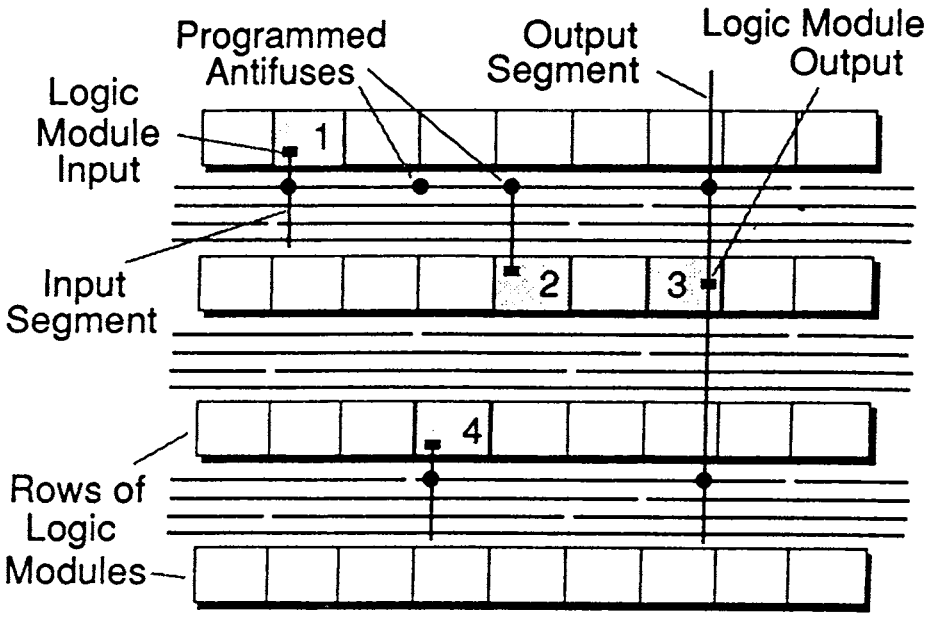


Figure 3.1.1. Actel Architecture

vertical wiring segment spanning either the channel just above or below the module. Each output signal appears on a dedicated vertical wiring segment of somewhat longer length. An antifuse is provided at each intersection of a horizontal and vertical segment, permitting them to be connected. The output of the driver, Module 3, in the figure, is connected to the Module 2 input by programmed antifuses to horizontal segments which, in turn, are connected to input segments. In the top channel, an antifuse is used to link two adjacent horizontal segments end-to-end, making it possible to reach an input of Module 1.

The central array of modules and channels is surrounded by input/output pads and buffers. Each IO buffer may be connected to the internal logic through a special module in an outer row or column of the array.

Succeeding sections of this chapter will consider the various factors that determine

the speed, cost and ease of use of antifuse-based FPGAs. Section 3.2 describes the antifuse device, including its physical structure, manufactureability and reliability. Section 3.3 explains some basic principles of programmable routing applicable to any programmable switch technology, the *segmented routing channel* model Actel's routing architecture, and the basic architecture of Act1, Act2, and Act3. Section 3.4 describes the design flow and the design automation tools. Section 3.5 summarizes how antifuse FPGAs are expected to evolve in the future. Finally Section 3.6 includes some design applications.

3.2 Programming Technology

The programming element is the key to an FPGA architecture. The characteristics of the programming element influences in many fundamental ways the viable architecture of the FPGAs. The requirements of a programmable interconnect switch for a high-performance FPGA include an element that has a small area and low parasitic resistance and capacitance, as well as a switch technology that is manufacturable and reliable.

Laser-programmed switches [Allen] [Smith] offer decent performance, but require costly equipment that must have direct access to the un-packaged die in order to program the part. Although the switch itself is small, it often requires a surrounding buffer zone to protect adjacent structures from being damaged by the laser. There are also significant programming time and yield considerations which reduce production viability.

Early PROMs and PLDs employed electrically programmed fuses made of such materials as polysilicon, platinum silicide, tungsten-titanium, and nickel-chrome. These materials have proven to be both difficult to manufacture and program reliably for integrated circuits. The most common difficulty is that a programmed fuse can *grow back, or reconnect* over time if it was not programmed with an adequate current.

More recent architectures have used transistors as interconnect switches [Carter]. Although this approach is widely used, it has some significant costs. The appreciable resistance and capacitance of the switch transistor, and the large area of the SRAM or EPROM cell controlling it constrain the design of the routing architecture and the performance of the circuit.

Actel architectures employ a one-time programmable element the size of a via called an *antifuse*. An antifuse switch technology offers the following advantages for FPGAs:

- Antifuses have a significantly lower *on* resistance and parasitic capacitance than switch transistors, reducing RC delays in the routing.
- Antifuses are small, typically the size of via, and sit at the intersection of the horizontal and vertical routing wires.

Antifuses are off devices in an unprogrammed state. Their size allows the use of simple and flexible routing architectures with 1,000,000 antifuses on a 14100 (10,000 gate) device. Only a small fraction of the total number of antifuses need to be programmed, about 2% for a typical application. Antifuses fall into two categories: amorphous silicon and dielectric.

A layer of amorphous silicon placed between two metal layers undergoes a phase change when current is passed through it, becoming conductive. Devices based on this principle have been the subject of research for many years [Gerzberg] [Whitten] [Roesner] [Holmberg] [Lim] [Stopper], and were considered for an early FPGA design[Graham]. Their use has been hampered by two difficulties. First, application of a reverse current can return the amorphous silicon in a programmed antifuse to a nonconductive state. Second, even unprogrammed devices pass a small but significant current, termed *leakage current.* In a memory, where only a few bits must be active simultaneously, the problems can be avoided by careful design. The problems are more significant in an FPGA since the supply voltage is present across about half the antifuses at any given time.

Recent efforts at developing an amorphous silicon antifuse for FPGAs report the following results. Resistance is inversely proportional to the programming current, and is 50-110 ohms with a mode of 80 ohms at a programming current above 10mA [Birkner]. The capacitance contributed by each antifuse is 1.3 femtofarads in a 1.0 micron CMOS process [Birkner]. (This number does not account for the capacitance of the metal lines themselves, which contribute several times this amount per antifuse.) Pre-programming leakage current is under 10 nanoamperes at 5.5 volts [Whitten].

Dielectric antifuse consist of a single or multi layer of dielectric material placed between N+ diffusion and polysilicon. Upon application of sufficient voltage, the dielectrics breaks down. Early dielectric antifuses used a single-layer oxide dielectric. The remainder of this section focuses on the Programmable Low Impedance Circuit Element (PLICE), a multi-layer oxide-nitride-oxide (ONO) dielectric antifuse developed for use in FPGAs [Hamdy]. The PLICE is small enough that the area of a switch array is limited by the pitch of the metal wires rather than the size of the antifuse itself.

In the unprogrammed state the PLICE has a resistance over 100 giga-ohm. The PLICE can be programmed in one millisecond by applying 16 volts across its terminals. This programming pulse melts the dielectric, creating a conductive link of polycrystalline silicon between the electrodes. Typically, a single link is observed. The radius of the link increases with programming current, hence lowering the resistance. As the programming current flows, dopant atoms flow from both electrodes into the link, providing a controllable low resistance.

For a minimal area PLICE programmed with 5mA, the resistance is distributed around 600 ohms as shown in Figure 3.2.1 The resistance distribution is significantly

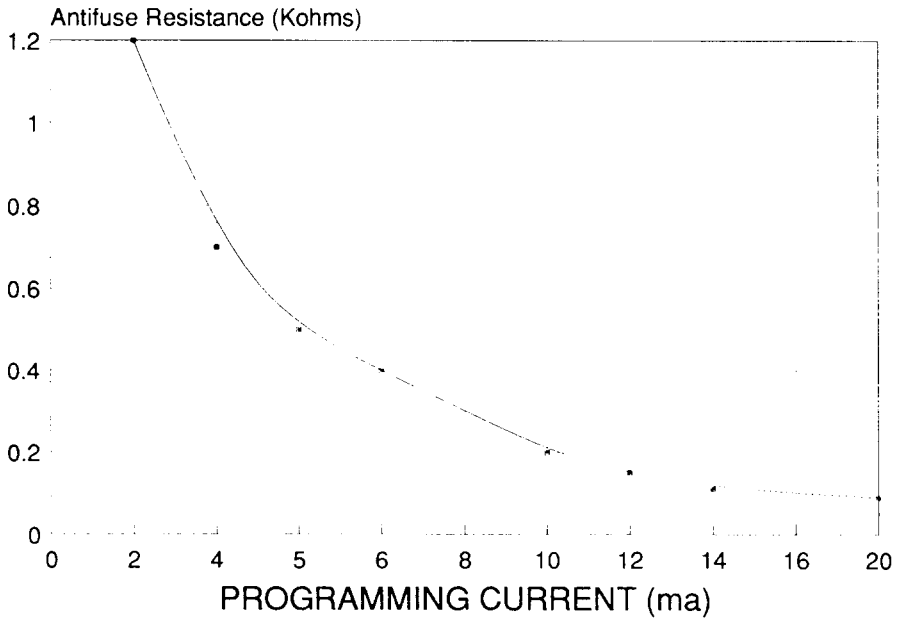


Figure 3.2.1. Resistance of Programmed Antifuse versus Programming Current

tighter than obtained with simple oxide dielectrics. With a larger cell layout that reduces the parasitic resistance to the electrodes, and higher programming currents of 14 mA, the distribution moves down to about 100 ohms, providing an area-delay trade-off. Use of ONO also improves both the yield and the reliability compared to oxide antifuses [Chiang1]. Capacitance is 6 femtofarads per antifuse in a 0.8 micron CMOS process; this includes the contribution of the polysilicon and diffusion electrodes and the metal lines used to connect them [Chen]. An unprogrammed PLICE has a leakage current of about one femtoampere, so even for the largest FPGAs the total leakage is negligible.

Use of the PLICE adds three masks to a conventional double-metal CMOS process. It can be fabricated in a typical CMOS facility using standard material, processing equipment and techniques.

A PLICE consists of an oxide-nitride-oxide structure sandwiched between N+ diffusion and N+ polysilicon gate, as shown in Figure 3.2.2 . A thin layer of oxide is thermally grown on top of the N+ surface, followed by LPCVD nitride and the reoxidized top oxide. Finally, the top polysilicon electrode is implanted with arsenic.

Antifuse reliability must be considered for both the unprogrammed and programmed

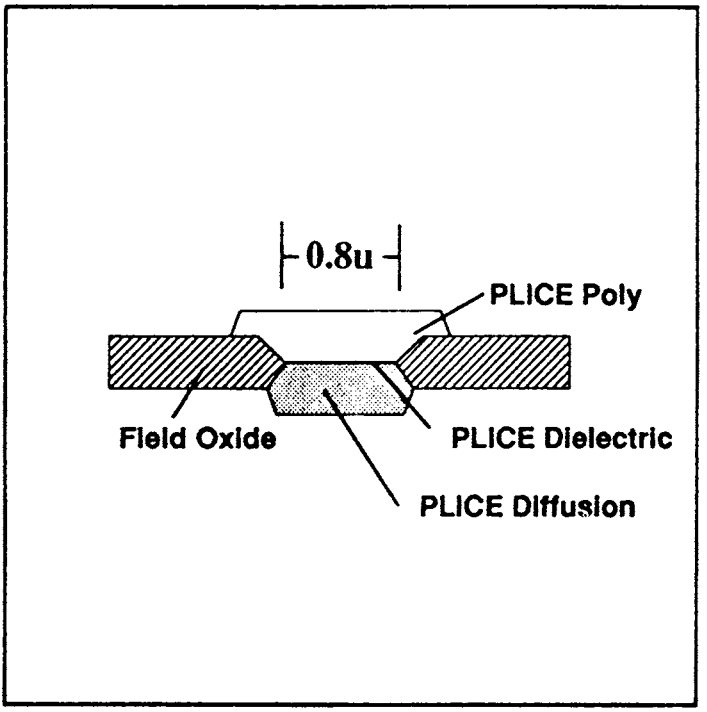


Figure 3.2.2. PLICE Cross Section

states. For an unprogrammed antifuse, with ONO less than 10nm thick, time dependent dielectric breakdown (TDDB) reliability time is an important consideration. Ordinary accelerated testing using electrical field and temperature stress was done in order to extrapolate the dielectric's lifetime under normal operating conditions. Figure 3.2.3 shows a plot of the time-to-breakdown vs. the reciprocal of the electric field applied to the dielectric, which has been shown to be an appropriate model [Chiang1]. Based on this data, one may extrapolate a lifetime for an ONO antifuse of well over 40 years of normal operation at 5.5V and 125C.

It is equally important that the resistance of a programmed antifuse remain low during the life of the part. Single-layer oxide dielectrics are known to be susceptible to *self healing* where resistance increases over time. Such increases do not occur for ONO dielectrics. Temperature-accelerated measurements reveal no intrinsic failure mechanism; the programmed antifuse resistance remains unchanged in all cases. The true lifetime of a programmed antifuse has yet to be determined since normal CMOS electromigration failures destroy the test structure first.

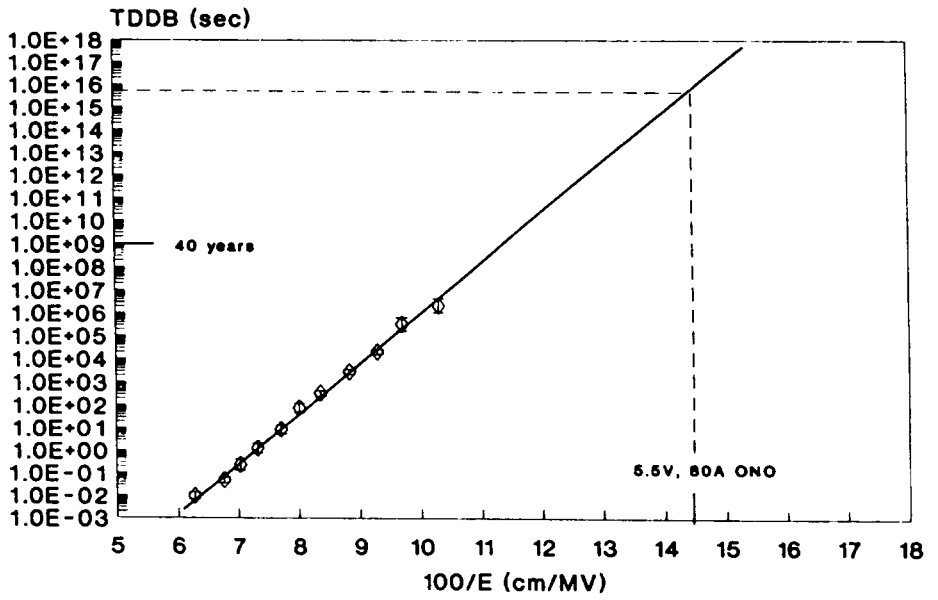


Figure 3.2.3. ONO Reliability, 1/E Model (95% Confidence Intervals)

Of course, the reliability of the FPGA is affected by the reliability of the base CMOS process as well as the PLICE. PLICE-based FPGA product reliability studies show the same failure levels encountered with normal CMOS circuits [Chiang2]. The ONO dielectric is highly radiation resistant. Initial results show that products containing ONO antifuses can withstand 1.5 million rads.

3.3 Device Architecture

This section describes in detail the Actel antifuse FPGA architecture. The first section is an overview of the architecture. Subsequent sections describe Act1, Act2 and Act3 focusing on their differences.

Principles of Programmable Routing

A routing architecture for an FPGA must meet two criteria: routability and speed. Routability refers to the adaptability of the wiring segments to accommodate all the nets of a wide variety of applications. Only the switches connecting the wiring

segments may be customized (by programming) for a specific application, not the numbers, lengths or locations of the wiring segments themselves. While sufficient wiring segments for good routability must be provided, excess wiring segments waste chip area.One of the important architectural considerations is that the routing of an application can be determined by an automatic routing program with little or no manual intervention required.

Propagation delay through the routing is the other major factor in FPGA performance. In any gate array architecture, whether mask or field programmable, it is inevitable that some nets require longer routings than others. After routing, the exact parasitic resistance and capacitance of the wiring are known and the delay on each net can be computed accordingly. The resulting *post-layout* net delays will vary according to some statistical distribution. If the average of this distribution is high, it will limit the performance of the design. If the distribution is too broad, a user will have difficulty estimating delays for the application.

The delay distribution for mask-programmed arrays is narrow from the low interconnection impedance so that variation isn't a major difficulty for the designer. The problem is more challenging for FPGA architectures. Any programmable switch (EPROM, MOS pass device, or antifuse) has a significant resistance and capacitance product (RC). Each time a signal passes through a programmable switch, another RC stage is added to the propagation path. For a fixed R and C, the propagation delay mounts quadratically with the number of RC stages in series. This increases the average of the net delays and broadens the post-layout delay distribution. The use of a low resistance switch, such as the antifuse, limits RC to keep the average delay low and its distribution tight.

Of equal significance to the performance advantages of the antifuse is optimization of the routing architecture that it affords. Some of the trade-offs between the length of wiring segments in a channel, the area required by the segments, and the resistance and capacitance of the switch are illustrated in Figure 3.3.1 .

Figure 3.3.1 (a) illustrates a set of nets routed in a conventional masked device channel. With the complete freedom to configure the wiring afforded by mask programming, the positions and lengths of the horizontal wires may be customized for the particular set of nets. The *left edge algorithm* [Hashimoto] shows how to do the customizing using a number of tracks equal to the *channel density.* The channel density is defined as the maximum number of nets passing through any cut across the channel [Lorenzetti]. This figure assumes there are no *vertical constraints* [Lorenzetti], since they do not occur in FPGAs where each signal enters or leaves the channel on its own vertical segment.

In an FPGA, achieving the freedom of an unconstrained channel would require switches at every cross point, as shown in Figure 3.3.1 (b). These switches between adjacent cross points along a track allows the track to be subdivided into segments of arbitrary length. Since the number of RC stages encountered by a net is proportional

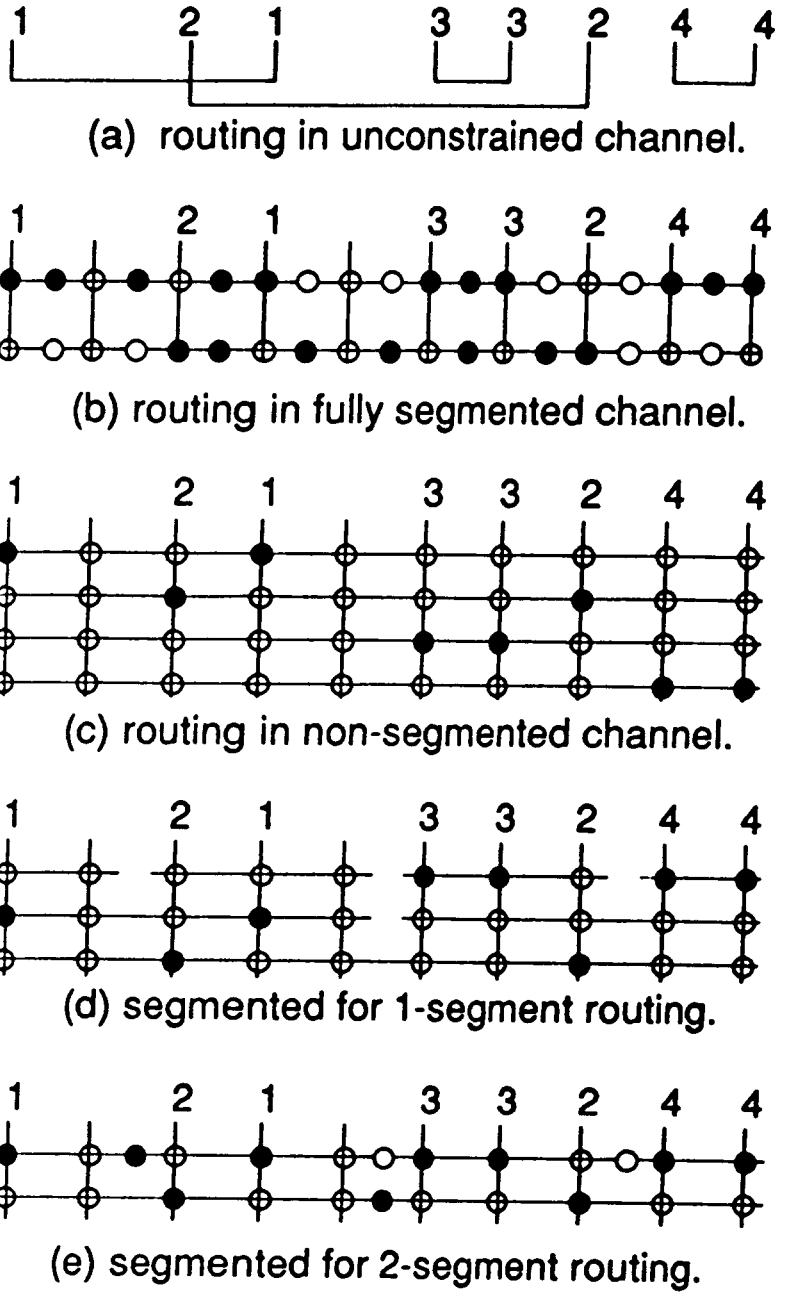


Figure 3.3.1. Illustration of Antifuse Routing

to its length, the delay of long nets becomes unacceptable.

Another alternative would be to provide a number of full length continuous tracks large enough to accommodate all nets, as shown in Figure 3.3.1 (c). This approach is used in the switch arrays of many types of programmable logic arrays and certain programmable logic devices (e.g. [Wong] [Marr]). The advantages of this model are that only two RC stages are encountered on any net, and that the delays of the nets are identical and predictable. However, even short nets incur the capacitance of a full track length. Furthermore, the area required is excessive and grows quadratically with the number of nets.

A segmented routing channel offers an intermediate approach. The tracks are divided into segments of varying lengths (Figure 3.3.1 (d)), allowing each net to be routed using a single segment of the appropriate size. Greater routing flexibility is obtained by allowing multiple adjacent segments in the same track to be joined end-to-end by switches (Figure 3.3.1 (e)). Enforcement by the software of simple limits on the number of segments joined or their total length guarantees that the delay will not be unduly increased by joining segments.

The problem of routing a segmented channel, or assigning a segment s to each unique net, is solvable in linear time for single-segment routing, Single-segment routing is illustrated by the model of Figure 3.3.1 (d). When a net is allowed to use multiple segments, as illustrated in Figure 3.3.1 (e), the general routing problem becomes more difficult (in particular it is NP-complete [ElGamal2]). However, many important special cases can be solved in polynomial time, and practical applications can be routed in a few minutes on a personal computer using heuristic methods. Reference [ElGamal2] gives a more thorough review of algorithms for segmented channel routing.

How does one design a segmented channel? In a conventional masked device channel the number of tracks, or *channel width*, must be chosen to accommodate most applications. Statistical models have been developed to estimate the required channel width (e.g. [ElGamal3]). In a segmented channel device, both the channel width and the segment lengths and offsets must be chosen carefully to suit the statistics of anticipated applications. Remember these applications are varied, and may have quite different routing requirements. Analytical solutions to the problem are as yet unknown, but experience indicates that even channels designed in an ad-hoc manner can be quite efficient (with limited use of multiple-segment routing).

Surprisingly, a well-designed segmented channel does not require many more tracks than would be needed in a conventional channel. It is an interesting finding, given the considerable restrictions segmented routing imposes, but it is supported both experimentally and analytically. A distribution giving the probability of occurrence of a connection as a function of length and starting point was derived from placements of 510 channels from 34 applications. Two segmentations were designed for a channel with 32 tracks: one intended for routing using exactly one horizontal segment

per net, the other for routing using one or two horizontal segments per net. Sets of nets with various densities were chosen randomly from the distribution, and an attempt was made to route each set in both channels.

The results of this test may be seen in Figure 3.3.2 . In a conventional channel any

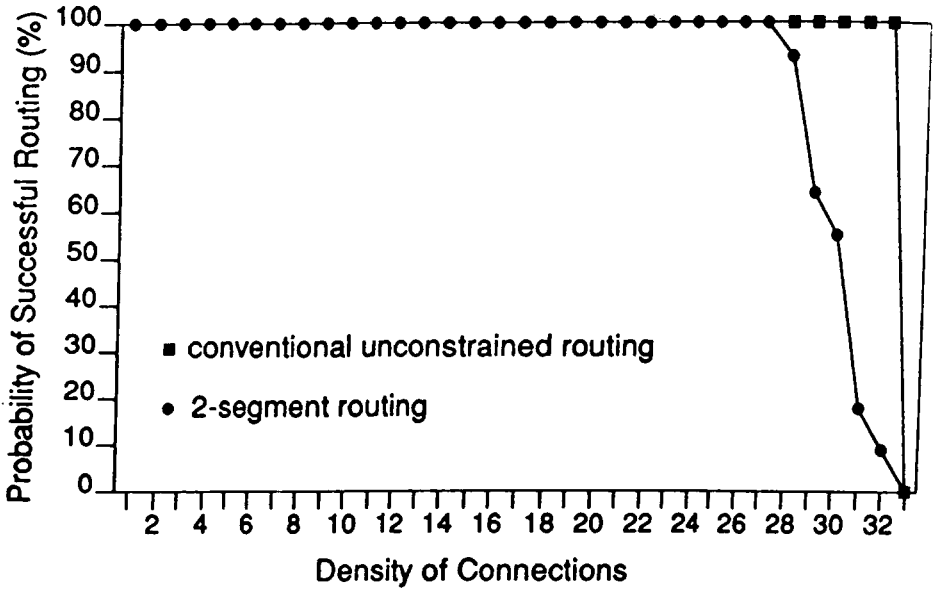


Figure 3.3.2. Results of Routing of 520 channels from 34 applications

design containing routing channels with density equal to 32 or less can always be routed. Allowing two-segment routing provides results only a little worse than the unconstrained case. A high probability of routing is observed when the density is only three or four below the number of tracks. (Further details of this study are given in [Greene1]).

An asymptotic analysis [ElGamal2] has confirmed that the number of tracks required in a segmented channel grows linearly with the expected channel density, just as with

conventional channels. This is true even for single-segment routing.

Routing Architecture of the Actel FPGAs

We now describe in detail how segmented routing is applied in the ACT FPGA routing architecture. This section presents a general view of the Actel architecture and is followed by sections describing Act1, Act2, and Act3 Figure 3.3.3 shows a

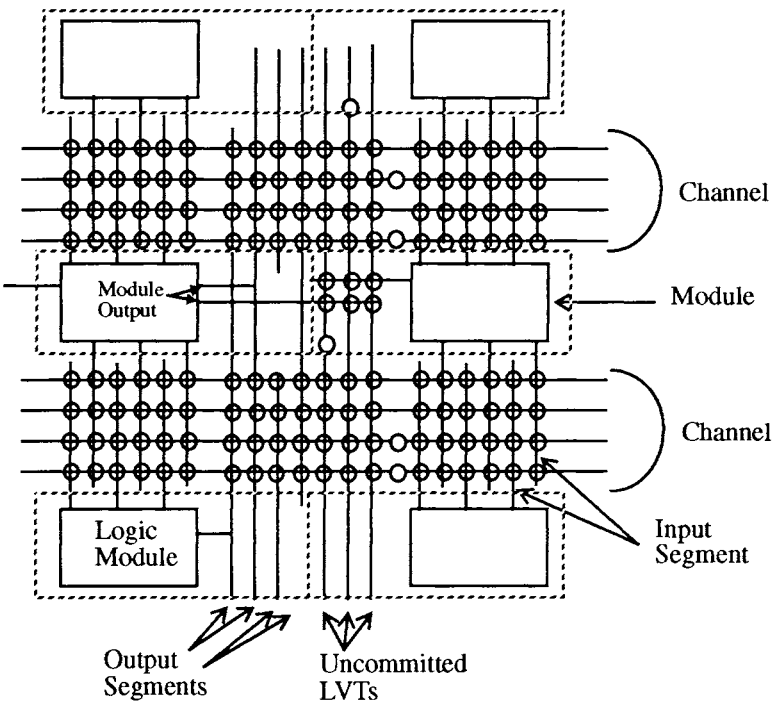


Figure 3.3.3. Simplified View of an Actel Module and Routing Segments

simplified schematic view of a module and the adjacent wiring segments and antifuses. Two segmented routing channels extend horizontally above and below the module. The input segments each span only one channel in order to minimize their capacitance and the total wiring area. Each input segment can be connected to any of the uncommitted horizontal segments in its channel through an antifuse. Other antifuses connect horizontal segments end- to-end to support multiple-segment

routing. Each channel also contains one full length segment that is grounded and another tied high so that any input can be programmed to logical 0 or 1. The figure shows six inputs per module, but the actual number varies according to the module function; it is typically eight for a basic logic module.

Output segments span two channels above and two channels below the module. Segments reaching the top or bottom channel may be slightly longer or shorter. Additional uncommitted vertical segments of varying lengths are also provided. They may also be joined end-to-end to form vertical segmented channels. (In the actual layout of the device all vertical segments pass over the modules and the term *channel* refers only to the region occupied by the horizontal segments.)

In Act2 and Act3 devices, a module output may also access the uncommitted vertical segments directly through one antifuse, called an *F antifuse,* outside the channel. Each module has access to the uncommitted segments on either of its sides allowing two modules to share the same set of *Long Vertical Tracks (LVT).*

The routing shown in Figure 3.1.1 is typical of most nets. In order to minimize the number of series RC stages, each channel's horizontal segments are driven directly by the dedicated output segment. This style of global routing is classified as a *Steiner Tree with Trunk.* [Preas].

Although the favorable routing of Figure 3.1.1 , can be assured for speed critical nets, some 5-10% of the other nets must be placed with a module input in some channel beyond the reach of the dedicated output segment. In this case, a suitable uncommitted vertical segment is selected to provide an alternate trunk for the channels not reached by the output. The vertical segment is driven directly from the output through an F antifuse, as shown in Figure 3.3.4 . Since this antifuse may be called upon to drive nearly the whole capacitive load of a widely dispersed net, the delay is very sensitive to its resistance. By programming these antifuses with higher than normal currents and providing them with additional strapping contacts their resistance is reduced to about 100 ohms, compared to the 500 ohms of a typical antifuse. The *F* antifuses thus require greater layout area but since there are few of them the cost is negligible.

In the rare instance where an uncommitted vertical segment is needed, but none is directly accessible to the module output, a segment from some other column must be used. In this case the uncommitted segment is driven through one of the horizontal channels spanning itself as well as the module output as shown in Figure 3.3.5 . The route involves an extra RC stage, so it is reserved for nets whose speed is not critical. Assignment of uncommitted segments in each horizontal and vertical channel to those nets that need them constitutes the segmented channel routing problem. Algorithms for solving these are discussed in [ElGamal2].

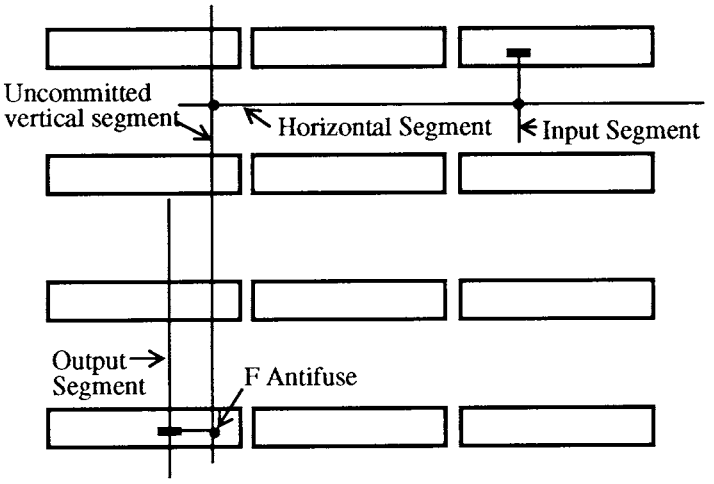


Figure 3.3.4. Routing Using Long Vertical Track (LVT)

Act1 Architecture

FPGA architectures can be characterized by the complexity or granularity of their basic logic module. A simpler module has lower internal delay and, since modules require less area, more of them can be provided on the chip. Furthermore, a *fine-grained* architecture tends to be more flexible. For example, a wide variety of functions may be built with equal efficiency out of two-input NAND gates, but eight-input NAND gates are much better at some functions than others. With simple modules there are also often more ways to implement a function allowing beneficial trade-offs between area and delay to be made.

On the other hand, using a module that is too simple can overburden the routing network. If a function that could be put into a few complex modules must instead be distributed among many simple modules, more connections must be made through the programmable routing network.

As a rule of thumb then, an FPGA should be as fine-grained as possible while maintaining good routability and routing delay for the given routing technology. The module should be chosen to implement a wide variety of functions efficiently, yet have minimum layout area and delay.

The approach used to select the Act1 modules was to employ macro usage statistics from masked gate array applications to evaluate candidate modules. This approach is

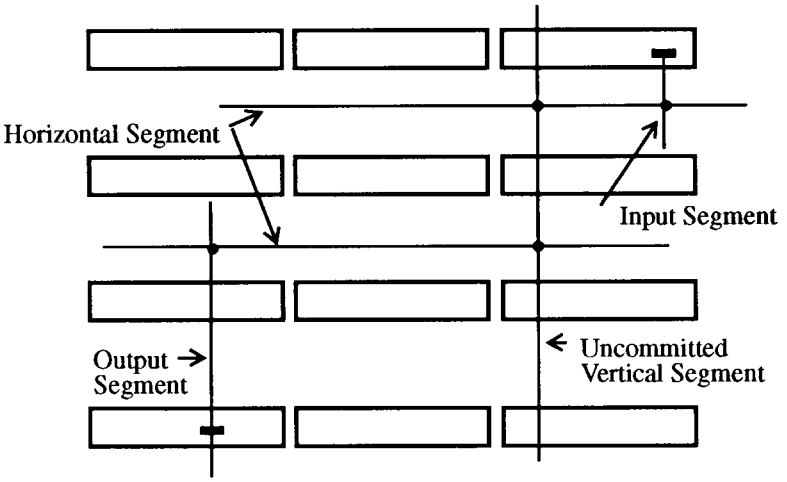


Figure 3.3.5. Routing Using LVTs in another Column

similar to that used to define the instruction set of a RISC microprocessor; choose a module that is most efficient for the most commonly used macros.

Given the parameters of antifuse routing, the effort focused on modules similar in complexity to a typical gate array macro. Statistics from designs targeted for implementation in a masked gate array were considered valid designs targeted at a candidate module. Only simple local optimizations (such as relocating inversions or combining a macro with a subsequent D flip-flop) influenced the module selection.

The Act1 family uses one general-purpose logic module [ElGamal1], shown in Figure 3.3.6 . Various macro functions (e.g. gates, flip-flops) can be implemented by applying input signals to the appropriate module input(s) and connecting other module inputs to logic 0 or 1. The module can implement any combinatorial function of two inputs, any function of three inputs (except the three-input NAND and exclusivity functions), many functions of four inputs, and other functions of up to eight inputs. Versions of AND gates and OR gates are implemented with all combinations of inverted and non-inverted inputs. In all, 702 distinct combinatorial macros are possible.

Any sequential macro can be configured from one or more modules using appropriate feedback routings. Over a range of designs, each module implements approximately 3.22 gates of logic; regardless of the ratio of combinational to sequential logic in the design.

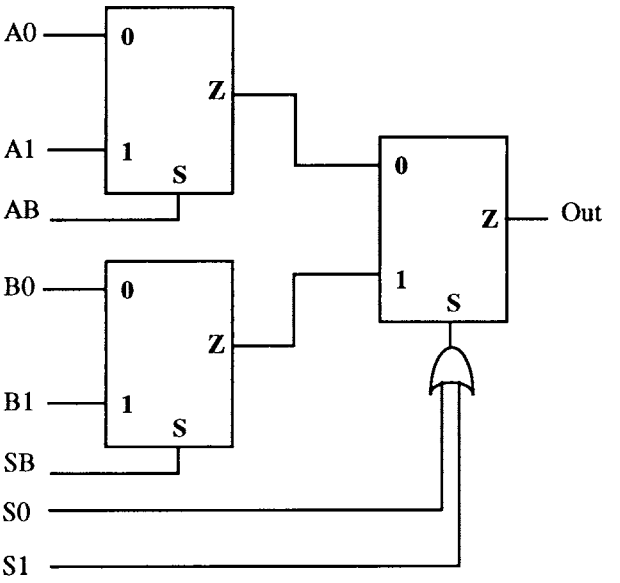


Figure 3.3.6. Act1 Logic Module

Each IO pad has an adjacent bidirectional buffer which connects to the array through an IO module. These modules are located in the outer columns and rows of the array next to the logic modules and interface to the routing channels in the same way as the logic modules. Each IO module has inputs for outgoing data and a three-state enable, and an output for incoming data which is driven from the pad. The IO module can be programmed to provide input, output, three-state or bidirectional capability.

The Act1 array is regular consisting of alternating rows of logic modules and routing channels, as shown in Figure 3.1.1 . The perimeter of the array includes IO modules. These modules connect the internal logic module signals to the pads. Each IO module has two inputs, data and enable, and an output. The data and enable signals are sent to the output buffer of the associated bonding pad, and the module's output comes from the input buffer of the pad. Thus the IO module can be configured to provide input, output, tristate, or bidirectional capability.

Clock signals present unusual requirements: they have high fanout and are required to have minimal delay and skew. Act1 devices meet the clock requirements with a dedicated clock network. Each network may be driven directly from an input pad for high speed. The signal passes through a buffer tree, and appears on a dedicated full-length horizontal track in each channel. Thus the network reaches every logic and IO

module in the array.

Act2 Architecture

Statistical indications that most of the nets driving the data input of a flip-flop have no other destinations motivated the selection of two specialized modules for the Act2 family. The Combinatorial (*C-module*) shown in Figure 3.3.7 is similar to the Act1

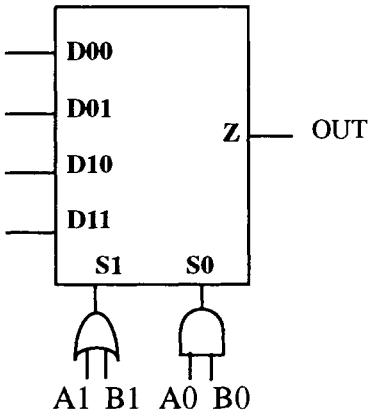


Figure 3.3.7. The Act2 Combinational Module (C-Module)

module. It was changed to better accommodate high fan-in combinational functions. It can implement 16 of the 20 four- input gates in the library, whereas the Act1 module implements eight, and some five-input AND and OR gates. The module modifications caused some loss in the ability to build sequential functions with C-modules. The module implements a total of 766 distinct combinational functions, including 13% more four-input and 12% more five-input macros than the Act1 module.

The Sequential module (*S-module*) consists of a front end equivalent to the C-Module followed by a sequential block built around two latches. Figure 3.3.8 gives a functional block diagram. The sequential block can be used as a rising- or falling-edge D flip-flop, or a transparent high or -low latch, by tying the C1 and C2 inputs to a clock signal, logical zero or logical one in various combinations. For example, tying C1 to 0 and clocking C2 implements a rising-edge D flip-flop. The block can also be set permanently transparent by tying C1 to 1 and C2 to 0, making the S-module

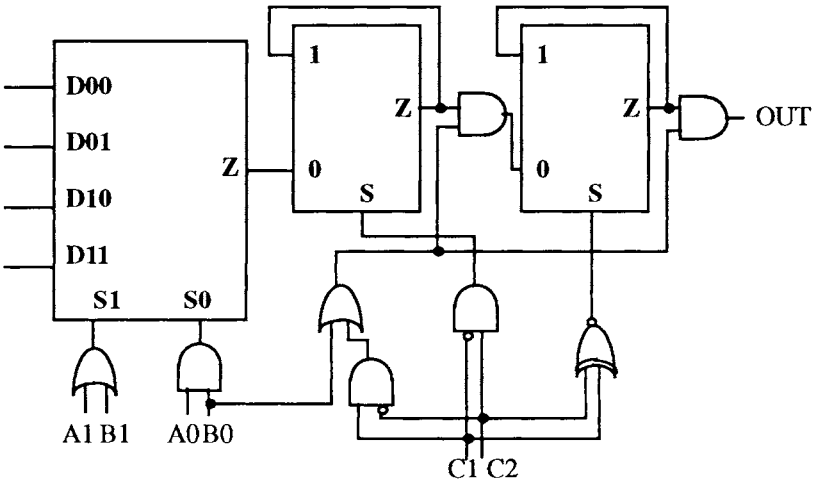


Figure 3.3.8. The Act2 Sequential Module (S-Module)

equivalent to a simple C-module with a small additional delay.

Figure 3.3.9 shows the functions that can be implemented using the S-module. Note that the latch with clear, a relatively rare macro, consumes part of the combinational logic to implement the clear; leaving a four-input function available in the C-Module. Toggle or enabled flip-flops can be made by using the logic in front of the D flip-flop. Other less commonly used flip-flops, such as JK or set/reset are not supported by the sequential block but may be configured from one or more C-Modules using external feedback connections as with the Act1 module.

A device with an equal mixture of C- and S-Modules provides a sufficient number of flip-flops for most designs with a margin to allow flexibility in placement. Over a range of designs, the Act2 modules provide about 1.4 to 2.0 times the logic capacity of Act1. As may be expected, the improvement is greatest for designs with a large proportion of flip-flops and wide gates.

Figure 3.3.10 shows a typical critical path in a state machine implemented with four C-modules and one S-module. The ability to fit a five-input gate in one C-module saves one routed net compared to the Act1 module implementation. The use of the S-module allows both the last combinatorial stage and the flip-flop to be combined into

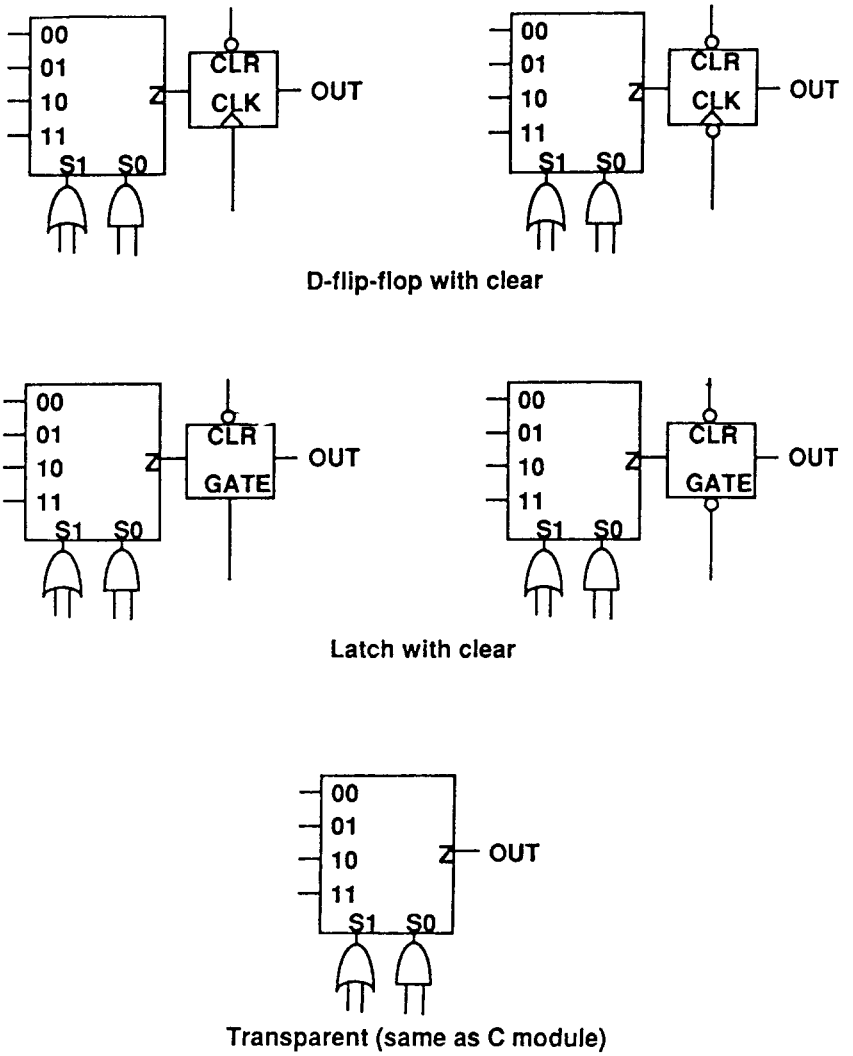


Figure 3.3.9. Act2 Sequential Module Configurations

a single module.

The choice of module also greatly influences the routability of designs. Because each input is accessible from only one of the two channels adjacent to a module, one might think that routability is worse relative to a conventional gate array cell, where a signal may enter from either channel. The impact of single channel accessibility, however, is not great because there is nearly always more than one way to implement a macro.

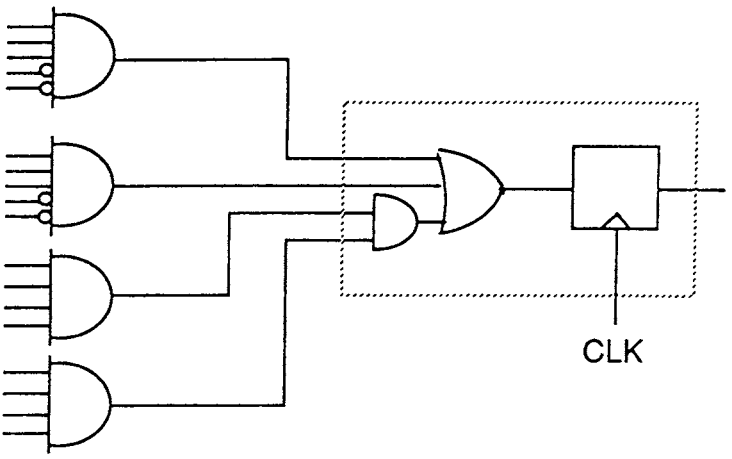


Figure 3.3.10. Act2 High Fan In Example

For an N-input macro there may be as many as 2^N different assignments of the input signals to the two channels. This corresponds to full double-entry symmetry. Even if not all of the 2^N assignments are possible, a sophisticated router can take advantage of implementation flexibility.

For the C-module, a given signal can be routed from either side of the module an average of 70 percent of the time. This number, weighted by macro usage, is quite sufficient for good routability. This figure is affected by both the module function and the assignment of a side to each module input. It is another important criterion in selecting a module function.

The module architecture of Act2 consists of alternating pairs of C-modules and S-modules. This provides the capability of implementing many macros that use pairs of combinational modules.

The Act2 family minimizes clock-to-output delay with a dedicated transparent-high latch in each pad, which can also be used as the slave stage of a flip-flop. The latch is controlled by a gate input to the IO module. If flow-through operation is desired, the gate is simply tied high to make the latch transparent. A dedicated transparent-low latch is provided on each input path. The polarities of the input and output latches are

chosen so they can be combined with each other, or with other internal latches or flip-flops, to form a chain of rising-edge flip-flops.

Clock signals present unusual requirements: they have high fanout and are required to have minimal delay and skew. Act2 devices meet the clock requirements with two dedicated clock networks. Act2 also provides the option to drive the clock network from user-defined internal logic as well as an input pad.

To minimize the capacitive load on the clock network on Act2 devices, antifuses are only provided to connect the clock track to certain module inputs, specifically the clock inputs of the S and IO modules and a subset of the combinational inputs on all the logic modules. Skew may be further reduced by having the automatic placement program attempt to balance the loading on each branch of the distribution tree.

Like other inputs, the clock inputs to the S-module, may be connected to the normal horizontal segments. This accommodates designs with several local asynchronous clocks. As with any technology, using asynchronous clocks with ACT FPGAs can cause race conditions and the designer must be alert to avoid them.

Act3 Architecture

The Act3 architecture is actually a twist on Act2, with key features added that optimizes the part for speed and flexibility. The Act3 logic module is very similar to Act2, in fact only two changes were made to the logic module, both targeted to increase flexibility and sequential system speed. The first change was to add a *clear* input to the sequential module, thus making the combinational portion of the sequential module completely independent from the sequential logic. There is a direct area cost to adding an input, but this addition provides a more uniform module for synthesis, and therefore increases the Actel flexibility as shown in Figure 3.3.11 . A user of the Actel array benefits greatly by being able to map more logic functions into the combinational portion of an S-Module. This aspect of the S-Module is called *combinable,* and is significantly better in Act3 over Act2. Figure 3.3.12 is a diagram of the S-Module, a C-Module corresponds to the Combinational block.

The other significant change in the Logic Module Architecture is the addition of a fast clock. This clock is not routed, and is significantly faster than its routed counterparts. The clock select, in Figure 3.3.12 , allows the selection between the high speed clock network, and the more flexible routed clocks.

Figure 3.3.11 illustrates the additional combinable logic functions available in the Act3 sequential logic module, as compared to Act2.

The other significant change in Act3 is the IO architecture. The Act3 is IO module rich, with flip-flops available on both the input and output paths. Figure 3.3.13 shows the IO Module. The IO module incorporates a separate fast IO clock. Synchronous enable signal for the flip-flops are also available as well as an output register feedback signal. This IO architecture is targeted for speed, and supports a

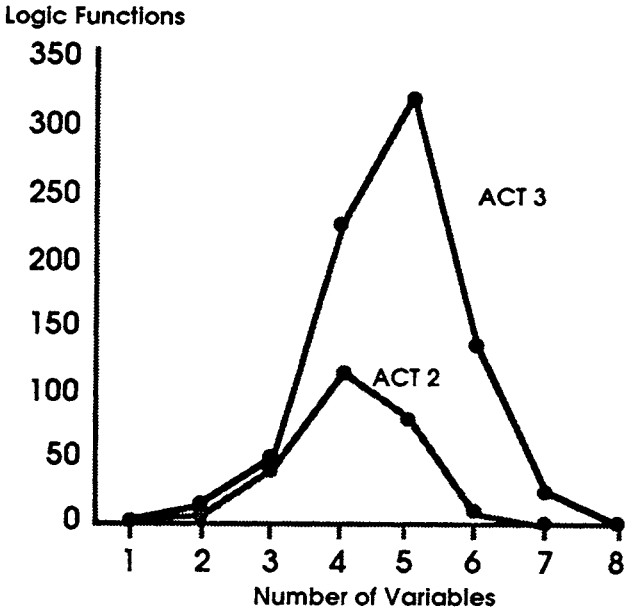


Figure 3.3.11. S-Module Combinatorial Logic Functions

significantly faster clock-to-Q, currently measured at 10 nanoseconds.

Programming and Testing

One of the great puzzles in the development of antifuse FPGAs was to find an efficient way to uniquely address each of the two-terminal programming devices. Diodes in series with each antifuse would allow unique addressing, but block signal flow. Early schemes required the use of an individual control line for each routing track, doubling the number of lines required in each direction [Graham]. Methods for programming and testing antifuse FPGAs that use only a few control lines for an entire channel were developed for ACT devices [ElGamal] [Ahrens]. Although a full description is beyond the scope of this text, the following explanation conveys the basic concepts.

Consider an array of antifuses at the intersection of some horizontal and vertical segments, as shown in Figure 3.3.14 . An antifuse is programmed by applying a programming voltage *Vpp* across it. This is done by precharging all segments to an intermediate voltage of about *Vpp/2*. Then a selected vertical segment is grounded and a selected horizontal segment is driven to Vpp. Other segments are left floating at Vpp/2. Only the single antifuse at the intersection of the selected segments sees the

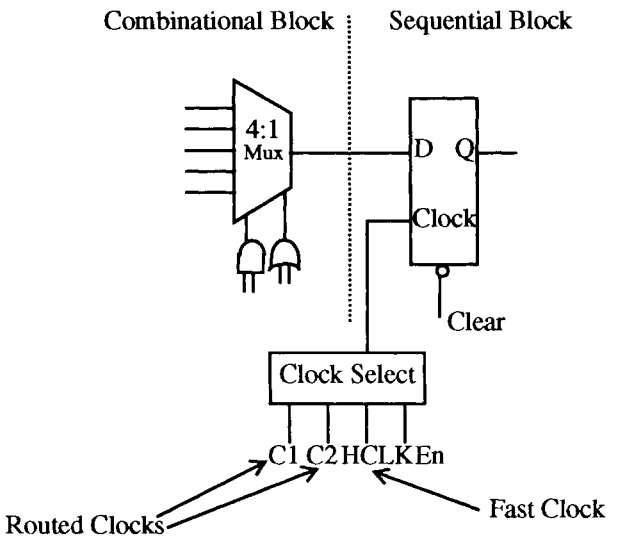


Figure 3.3.12. The Act3 Sequential Module

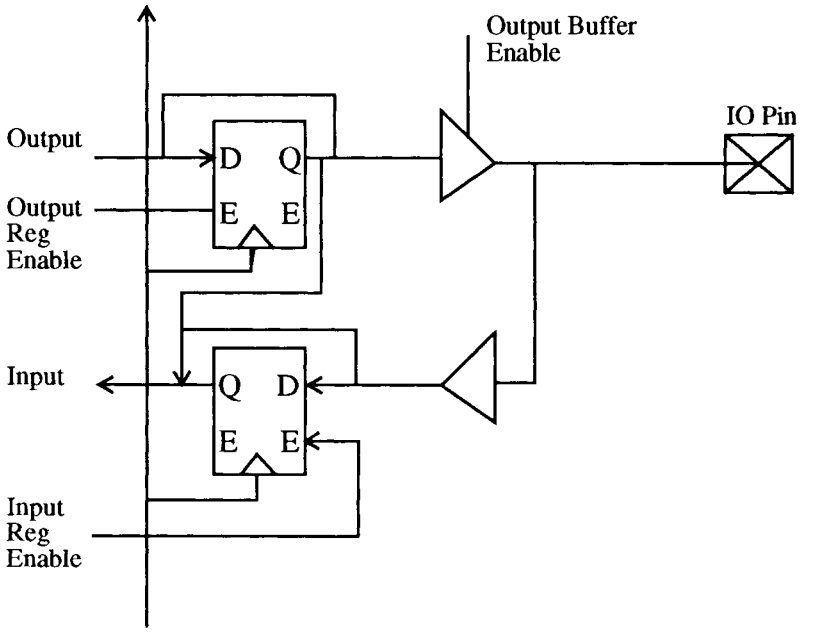


Figure 3.3.13. Act3 IO Module

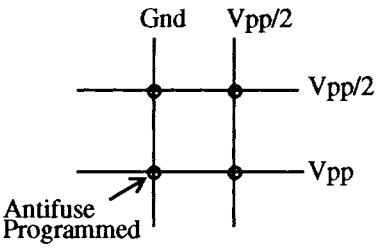


Figure 3.3.14. Programming an Antifuse

full Vpp.

Only the unique antifuse with the full Vpp potential across it is programmed. Figure 3.3.15 illustrates the *series pass transistor addressing* method. The structure is similar to that of Figure 3.3.3 with the modules removed for clarity. Each pair of adjacent segments in the same horizontal track is connected by a pass transistor. In tracks containing uncommitted segments, there is an antifuse present between the two segments, and so the transistor is connected in parallel with the antifuse.These transistors are used only for programming and testing and are shut off during normal operation of the programmed part. The transistors in each row of modules are gated by a control line, marked as *Row Control Line* in the figure, and the transistors in each column of modules are gated by a control line, also marked in the figure. These lines are driven, in turn, by programming logic present at the array periphery. Note that only one control line per row or column is required, regardless of the number of tracks.

Figure 3.3.16 illustrate the circuitry used to program an antifuse at the intersection of a horizontal and vertical segment. The vertical track containing the antifuse is driven from one end to ground by the programming logic present at the array periphery. The pass transistors in all rows between the driving periphery and the antifuse are turned on. The horizontal track is driven to Vpp as shown in the figure. A fuse between two adjacent segments in the same track is programmed in a similar manner. All columns of pass transistors except the one bypassing the antifuse are turned on, and the track is grounded at one end and driven to Vpp at the opposite end.

Figure 3.3.17 illustrates *direct addressing* circuitry, which is some cases offers a favorable alternative to series pass transistors. Column voltage supply lines run vertically through the array. Each segment along a track may be connected through its own addressing transistor to a supply line. These transistors are gated by a horizontal

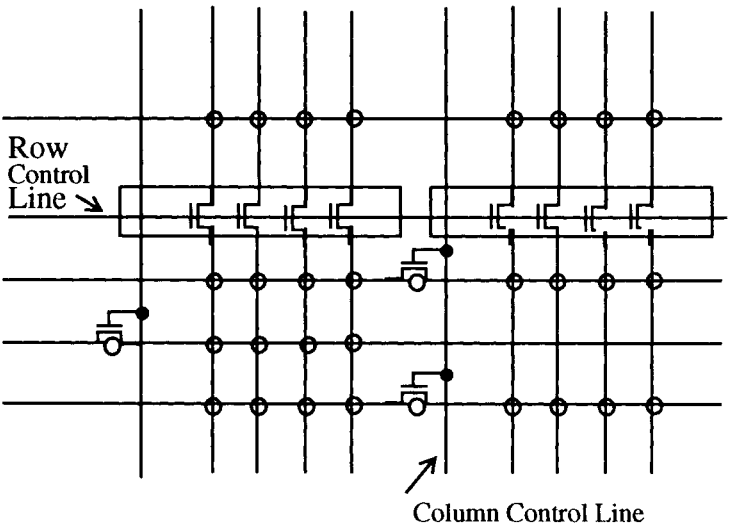


Figure 3.3.15. Antifuse NetWork

select line. Each select line serves all the segments in a group of one or more adjacent tracks in the same channel. Activating one vertical supply line and one horizontal select line uniquely addresses a particular horizontal segment. The number of segments in a channel that can be addressed is limited by the number of supply lines times the number of group select lines. It follows from this that the ratio of tracks to select lines is at most the average segment length. An antifuse between two adjacent segments may be programmed by activating the control line for both segments and two supply lines, one for each segment. Since only one transistor lies between a segment and the supply, the programming current is independent of the position of the segment and the segmentation of its track. Thus the direct address method is most efficient for irregular channels with long segments.

Act1 devices used only the pass transistor method. Act2 and Act3 devices use the pass transistor method for the vertical tracks, which contain many short input segments, and the direct address method for the horizontal tracks.

In either scheme, some care is required to assure that a unique antifuse is addressed once other antifuses have already been programmed. Figure 3.3.18 provides an

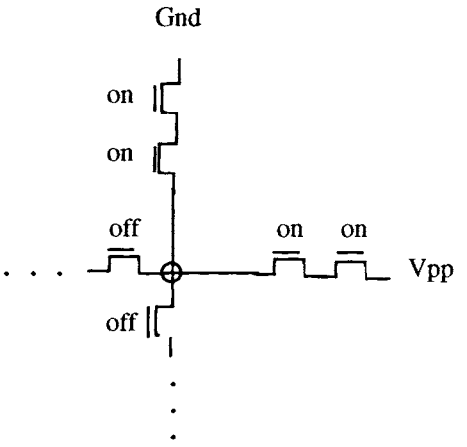


Figure 3.3.16. Antifuse Programming Example

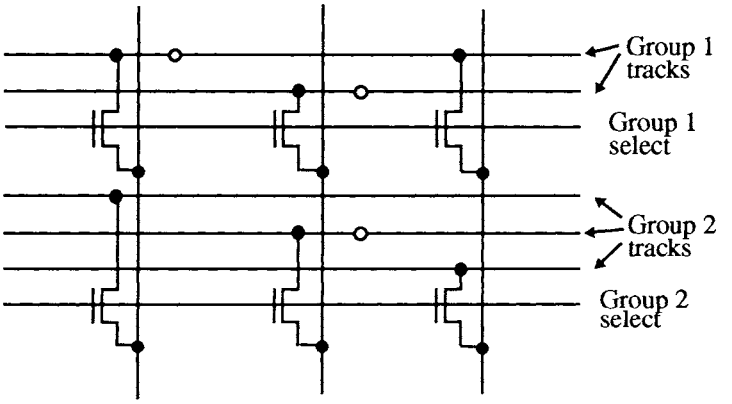


Figure 3.3.17.Direct Addressing Of Fuses

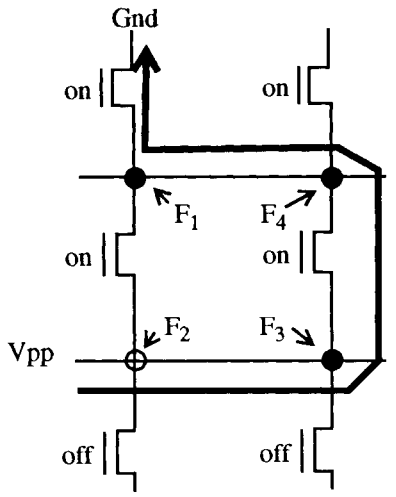


Figure 3.3.18. Programming Sneak Path

example of how improper addressing allows programming current to divert along a *sneak path*. The previously programmed antifuses F_3 and F_4, cause the programming of antifuse F_1 instead of the intended antifuse F_2. Unlike PROMs, FPGA programming does not involve an arbitrary pattern of antifuses. For example, it is not necessary to program a pattern connecting two outputs together since this does not form a useful net. For the relevant patterns, it may be shown that there is always an order in which the antifuses may be programmed with no chance of a sneak path occurring.

Special care is also required to protect the module circuitry from the voltages present on the segments during programming. Transistors that are in contact with the routing segments must be designed to withstand to the programming voltage.

Device testing takes place in three phases: before, during and after programming. Pre-programming tests check for shorted or open segments, shorted or even relatively low resistance antifuses, and proper module and IO operation. Continuity of the segments is easily tested by turning on all pass transistors and using the peripheral circuits to drive the segment at one end of a track and read the segment at the other end. Testing for the absence of shorts between segments in adjacent tracks is done similarly by applying a pattern of alternating zeros and ones. Weak antifuses may be screened out by applying the proper stress voltage (higher than normal operating voltage but lower

than Vpp) across groups of antifuses in parallel using the programming circuits. Breakdown of an antifuse is detected by passage of excessive current through it.

To verify the functionality of the modules we need to apply test vectors to their inputs and read their outputs. A vector may be applied simultaneously to an entire row of modules by turning on all vertical pass transistors except those in the row being tested. Data are applied to the inputs in the channel above the row from the top periphery and to the inputs in the channel below the row from the bottom. A simple row-select and column-sense scheme conveys the output of each module in turn to an output pad for monitoring.

Proper closure of the programmed antifuses is verified during programming by sensing the passage of programming current. A complete test for unintended connections between any two segments can be done after programming using the programming circuitry to precharge, drive, and read the segments. This is true despite the fact that it is no longer possible to uniquely address each individual antifuse once programming commences. The reason is that detection of shorts, but not the location, may be accomplished simultaneously for many antifuses in parallel. Taken together, the tests described insure correct and complete functioning of the programmed part.

Capacity

Determining the capacity of FPGAs is a complex task with many considerations. ACT device capacities are specified in terms of masked device gates. Masked gate arrays are specified to contain a number of two-input NAND gates. Utilization of the gates is a measure of the percentage of the total number of gates on the device that a typical design may use before running out of routing resources.

In an FPGA there are two utilization questions: how many of the modules can be placed and routed, and how many gates are obtained per module. ACT devices have a small logic module with the low granularity of a gate array as well as sufficient routing resources to achieve high module utilization. The module utilization of ACT devices is guaranteed to be 85% or greater with many designs achieving utilizations over 95%.

Both Act1 and Act2 devices have been tested for capacity using four well-known benchmarks. The results appear in Table 5 . Customer designs have also been used to analyze Act1 module gate utilizations. On average, Act1 modules implement 3.22 gates per module for all types of logic. The mixture of gates to flip-flops in a design does not affect the gate utilization of Act1 devices.

Table 5: Benchmark Capacity Results

Device	Benchmark	Gates	Instances	Total Gates
A1020	Data Path	157	12	1884
	Timer/ Counter	248	6	1488
	State Machine	153	9	1377
	Arithmetic	295	6	1770
Average				1630
A1280	Data Path	157	54	8478
	Timer/ Counter	248	26	6448
	State Machine	153	37	5661
	Arithmetic	295	18	5310
Average				6474

Table 6 summarizes results for a variety of designs in the A1280 and A1240 FPGAs. The designs were placed and routed automatically with no manual intervention. For this reason, the table includes the number of routed pins per logic module as a measure of design complexity. Nearly all designs with module utilization under 85%, most designs with utilization under 95%, and many with utilization up to 100%, route completely.

Table 6: Capacity Benchmarks

1280 FPGA 1232 Logic Modules	Logic Module Utilization	Pins Per Logic Module
Design done in an 8K TI Gate Array	99.5%	4.28
32 Bit Data Path, 16X16 Mult., State Machine	99.4	4.30
2901 ALU (X4)	98.1	4.57
DMA Controller (X3)	97.1	3.99
Asynchronous Serial ECC	97.0	4.78
Pipelined Fixed Point Mult, Div, Sqrt	94.5	3.37
State Machine, Mult/Add, Datapath, Counter	92.7	4.34
Color CRT Controller (X3)	87.3	3.83
32 Bit Datapath with Sum, Compare (X3)	86.8	5.25
40 Bit FLoating Point Adder/Subtractor	86.7	4.33
1240 FPGA 649 Logic Modules	Logic Module Utilization	Pins Per Logic Module
16 Bit Datapath, 16X16 Mult., State Machine	98.1	4.86
2901 (X2)	93.2	4.68
DRAM, DMA and SCSI Controllers, UART	92.6	4.73

Performance

Like capacity, performance measurements of FPGAs is problematical. Specifications such as maximum clock frequency or module propagation delay are insufficient indicators for estimating the performance of a given design. Like masked devices, the propagation delay of FPGAs is routing and fanout dependent. The ACT databook gives numbers for propagation delay based on device characterization for various fanouts.

Figure 3.3.19 shows the cumulative distribution for the routing delay to each input pin for various ranges of net fanout. The histogram includes all nets in the design. Pins on nets identified by the user as speed critical would generally be among the fastest 30%, and nearly always among the fastest 90%. The routing delay distributions are reasonably tight, especially for low fanouts and critical nets.

Table 7 shows results from tests on some typical circuits. The table shows the measured performance for the Act2 1240-2 and the Act3 1425, as of June 1993.

Table 7: Performance Benchmarks

Example	Speed (MHz)
Act2 1240-2	
16 bit counter	58
16 bit accumulator	36
state machine	39
Act3 1425	
16 bit counter	68
16 bit accumulator	40
state machine	46

The Act3 architecture is specifically targeted for speed. The measured on-chip performance is 125 MHz with 10 ns clock-to-out. Worst case performance benchmarks show 40 MHz 24-bit accumulators, 75 MHz 24-bit loadable counters, 125 MHz 24-bit pre-scaled loadable counters, 125 MHz shift registers, and 80 Mhz chip-to-chip speeds.

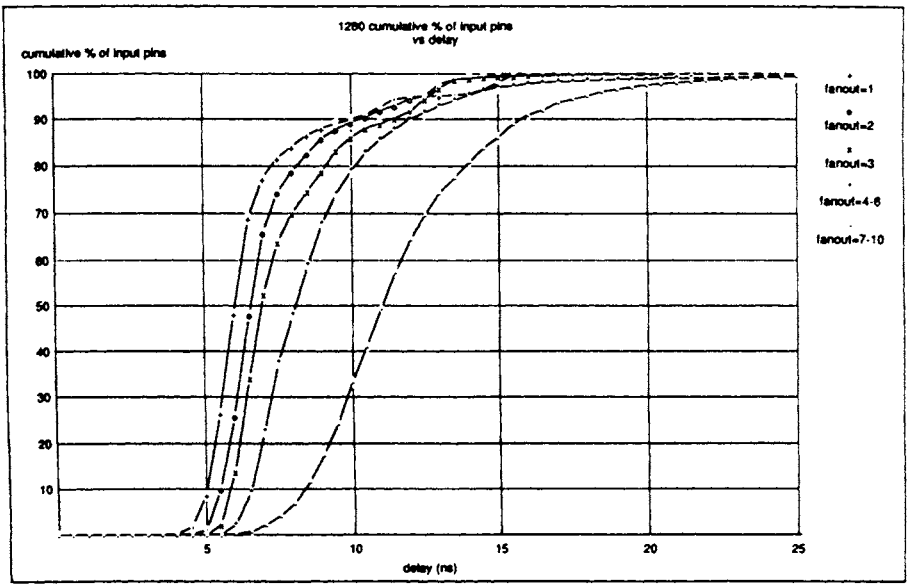


Figure 3.3.19. Cumulative Percentage of Input Pins Versus Delay

3.4 Software

Figure 3.4.1 diagrams the process of embedding a design in an Actel FPGA. This process begins with capture of the design in a computer readable format. Currently most users enter their designs as schematics built of macros from a library. Any of several standard schematic capture programs can be used. Since the Act1 and C-modules have the same complexity as a typical gate array library macro, the problem of selecting which groups of macros should share one module is avoided; each macro is typically assigned its own module. The larger capabilities of the S-module are handled by software that automatically combines a flip-flop and a preceding combinational macro into one S-module where possible. This process is transparent to the user and does not require modifying the schematic.

Another increasingly popular alternative is to enter designs in terms of Boolean equations, state machine descriptions, or functional (rather than structural) schematics. Different portions of a design can be described in different ways, compiled separately and the results merged according to a top-level hierarchical

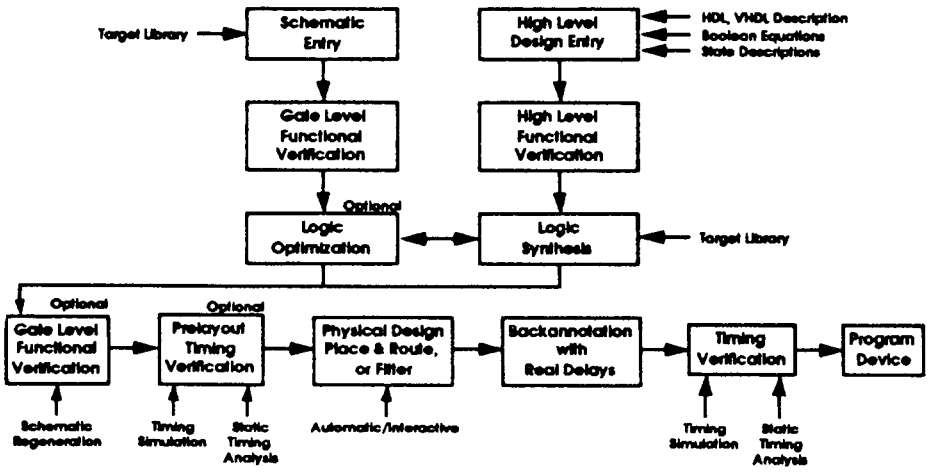


Figure 3.4.1. FPGA Design Flow

schematic. Various industry standard formats are supported.

High-level synthesis tools such as MIS-II [Brayton] or Synopsys [Rudell] can be used with Actel FPGAs by providing a suitable library. These tools can either encompass only the standard schematic library macros, or include the larger set of all 700 functions embeddable in the typical Actel module. Recent research has investigated other more efficient ways to allow synthesis tools access to the full flexibility of the module. A method for rapid searching of libraries using Boolean matching is given in [Mailhot]. Other approaches take advantage of the multiplexor structure of the module, using either binary decision diagrams [Murgai] or *if-then-else DAGs* [Karplus].

Several guidelines are suggested for reliable logic design with FPGAs. The general goal is to make proper circuit function independent of shifts in timing from one part to the next.

- Use synchronous logic design where possible.
- Avoid gated clocks, using enabled flip-flops instead.

- Avoid race conditions.
- Limit fanout by buffering.

These guidelines are the same as those for users of mask programmed gate arrays [LSI]. In addition, designs for Actel FPGAs must use multiplexers rather than tristate drivers since internal tristate is not supported.

Once the design has been entered it may be simulated either functionally or using pre-layout delay estimates before entering the ALS system. ALS provides for the selection of a device, package before the netlist is checked by a validation program for problems such as undriven nets, outputs shorted together, excessive fanout, etc.

A good pin assignment is key since it can impact the design's use of routing resources. Pin assignment may be done manually, or by the software, which bases its assignments on an analysis of the design.

The next step is to map the netlist into the Actel architecture. The placement problem, or selecting a module for each macro, is similar to that of a conventional gate array. Since macro placement effects all aspects of the design, including its speed and the number of gates used, there are a number of different placement options available. Automatic placement used algorithms similar to gate array programs, with a few additional considerations such as fixed power and programming pins. Incremental placement, as shown in Figure 3.4.2 , takes a netlist that has been modified slightly and updates the current placement slightly. Incremental placement is useful because it maintains similar net delays for subsequent placement runs.

Since the key is ensuring that critical nets meet speed requirements, another desirable alternative is Timing Driven Placement. Such a program performs placement based on timing constraints on speed critical paths. Current research suggests that definite improvements in routing delays can be made by applying these techniques.

Routing, described at length in "Principles of Programmable Routing" in section 3.3, is the next step. Routing congestion within each horizontal or vertical channel must be limited. Whenever possible, the dedicated output segments should be used in preference to the uncommitted vertical segments, especially for nets identified by the user as speed critical. Routing of the segmented channels is done as previously described. On Act2 devices, macros must be placed in modules of the appropriate type (C or S). Time for complete placement and routing of an 8000-gate A1280 FPGA is about 30-40 minutes on a 486.

Once placement and routing are completed, the propagation delay to each input pin is calculated based on device characterization data. The calculation accounts for the module internal delay as well as the delay through each RC stage in the routed net. The estimates can approach the accuracy of a SPICE circuit simulation. The delays may be back- annotated to a simulator, or checked with a static timing analyzer. In either case, timing data may cause an iteration in the design. At some point the design

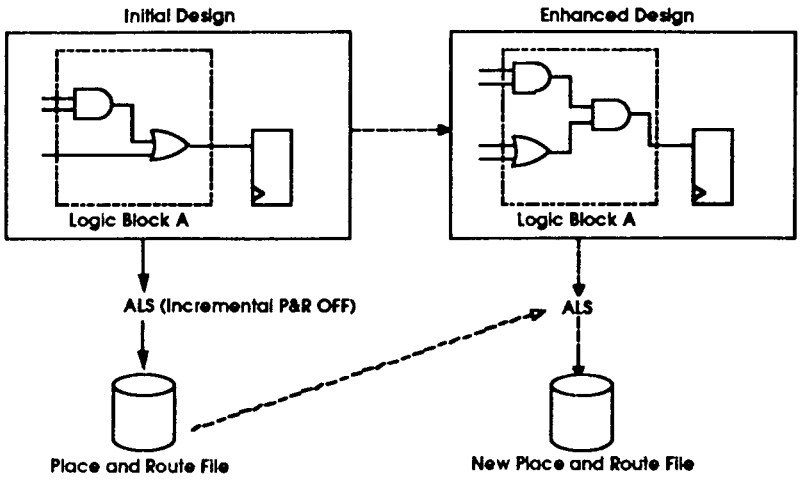


Figure 3.4.2. Incremental Placement Design Flow

meets whatever criteria the user is targeting, and it is time to program the FPGA part.

A list of antifuses to be programmed is generated and downloaded to a programming station which also holds the FPGA. Programming time is about 5-10 minutes, depending on the size of the FPGA and the design. The Actel Activator2 station allows up to four chips to be programmed simultaneously. The standard tests applied before, during and after programming, described in section 3.3 assure correct implementation of the programmed part. No design-specific test vectors are required from the user.

The programming system can access the test circuitry inside the programmed FPGA allowing it to sense the value of any pair of selected module outputs and present them on an external pin in real time. This unique probe feature, called the debugger, can be used as a debugging tool for the device while it is operating at speed in the system. The probe provides a useful back-up when simulation is difficult or impossible, such as for highly asynchronous designs or when the design must accommodate a poorly documented interface. Debugger software is also provided to enable the programmer to be used as a functional tester, presenting stimuli at the FPGA's pins and reading data back via the probe and the pins. Figure 3.4.3 illustrates the Actel design system.

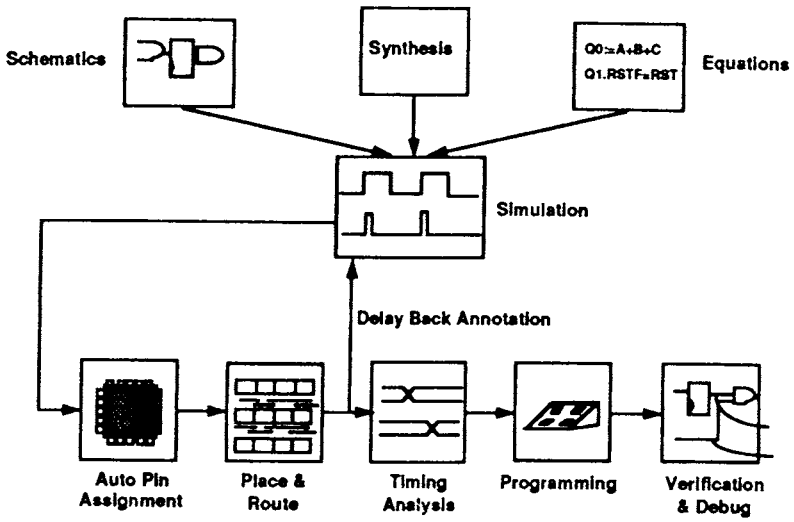


Figure 3.4.3. The Actel Design System.

3.5 The Future

How will the speed, density and cost of antifuse FPGAs evolve? Antifuse FPGAs will directly benefit from anticipated improvements in the underlying CMOS technology. In addition, it is likely that further improvements will be made in the antifuse itself. Changes that reduce the resistance, capacitance, or programming voltage would be beneficial and could significantly benefit architecture possibilities. On the architecture front, better logic modules and improved methods of designing segmented routing channels may be developed.

Antifuse technology provide the opportunity of developing sea of module (SOM) architectures, unlike other FPGA technologies. Just as gate array architectures placed the routing channels over the modules, the antifuse based channeled architecture has the same potential. A sea of module architecture can reduce the die area by up to half of the array size. Current process technologies have the promise of being able to place the antifuse on top of the logic modules.

3.6 Design Applications

Designing with ACT1 and ACT2 FPGAs

The advantages of using FPGAs over discreet devices are well known. In order to maximize the benefits of high integration and low power, it is important for the designer to understand how applications are mapped into the architecture, and trade-offs in applications. It is up to the designer to understand how to select components from the library so as to take best advantage of the logic modules. While it is not required to understand the architecture of Actel FPGAs to implement designs with them, there are architectural considerations that would aid designers if they kept them in mind when entering a design. Some considerations apply to all Actel devices, others are architecture specific. The differences between Act1, Act2 and Act3 devices also suggest some differences in the design considerations for using each of the families of devices. The applications in this section use Act1 and Act2. The Act3 applications are similar to Act2 only faster.

In order to take best advantage of the logic integration capability of an FPGA, designs that target such devices should implement functions using the fewest number of logic modules possible. The issue of capacity is closely related to that of efficiency. It also relates to performance. Sometimes the fastest design is also the smallest. In other cases using parallel logic increases the module count, but improves performances by reducing the number of logic levels.

An efficient implementation is one that uses the fewest number of logic modules to implement as many equivalent gates as possible. An inverter would be a very inefficient use of an Actel module while a single-bubbled input AND-EXOR (equivalent to 4.5 two-input NAND gates) would be very efficient.

Inverters or other library cells that are inefficient should be avoided. In most cases bubbled inputs on macros are available to implement inversions at no cost in modules. Two-level cells (e.g. AND-OR) are advantageous from both a capacity and performance perspective. Such cells compress logic to reduce delays and increase the number of gates per module.

Performance

The performance of logic is variable, so it is important to optimize designs. There are four criteria that influence performance of logic in FPGA designs. They are listed below in descending order of importance:

- **Levels of Logic** The fewer number of combinatorial logic levels between flip-flops the faster the logic.
- **Fan-out** Propagation delays in FPGAs are sensitive to fan-out. Limiting fan-out on individual nets improves performance.

- **Fan-in** Measures the number of nets connected to a logic module's inputs. Library functions with heavy fan-in efficiently utilize the logic of the module and aren't a problem when used sparingly. Too many high fan-in macros, however, can congest routing and reduce performance.
- **Number of modules** Fewer logic modules allow them to be placed closer to each other. Shorter distances between modules speeds the connection paths.

Chip Level Design Considerations

While each of the above considerations is important in itself, it must be remembered that they are interrelated and an improvement in one may cause a degradation in another. For example, limiting the number of logic levels tends to increase fan-in. If the design has a significant number of high fan-in macros, use of additional high fan-in macros in a counter may cause routing congestion. A balance must be found among them so than none becomes a drag on performance. The optimal solution to a design may only be found after some iterations that adjust for such things as fan-out and criticality. The ALS tools such as the *Validator* and automatic Place and Route can rapidly provide answers to questions about design capacity and routability.

Act1

The multiplexor-based logic module is highly efficient in multiplexor implementations allowing versions of both 2:1 and 4:1 to be implemented in one module. The Act1 library contains all bubbled input permutations of AND, OR, NAND, and NOR gates. All two-input and virtually all three-input gates are configured from a single module.

Eight of the twenty four-input gates offered require one module while the rest use two modules. Using the single module gates saves module resources and may be done by matching bubbled driver outputs with bubbled gate inputs as shown in Figure 3.6.1 . When a two-module gate is required, the delay penalty is not as great as two levels of gating because the two modules of the macro must be placed adjacent to one another.

Many highly efficient multi-gate macros are available in the Act1 library. They include AND-OR, OR-AND, and AND-EXOR functions. Designers should become familiar with the library so they can make the best use logic resources on the device.

A single Act1 module can be used as a latch, while flip-flops require two modules. Latches should be used whenever the design allows it. When flip-flops are required you should use those that the highest number of gates for the modules. There are several enabled and multiplexed flip-flops available in the library.

A flip-flop enable can be used in place of an AND gate driving the data input. The two-input multiplexor on a flip-flop can be used as a multiplexor or it may be used as a gate by tying one of the multiplexor inputs to a rail. The possible configurations are shown in Figure 3.6.2 .

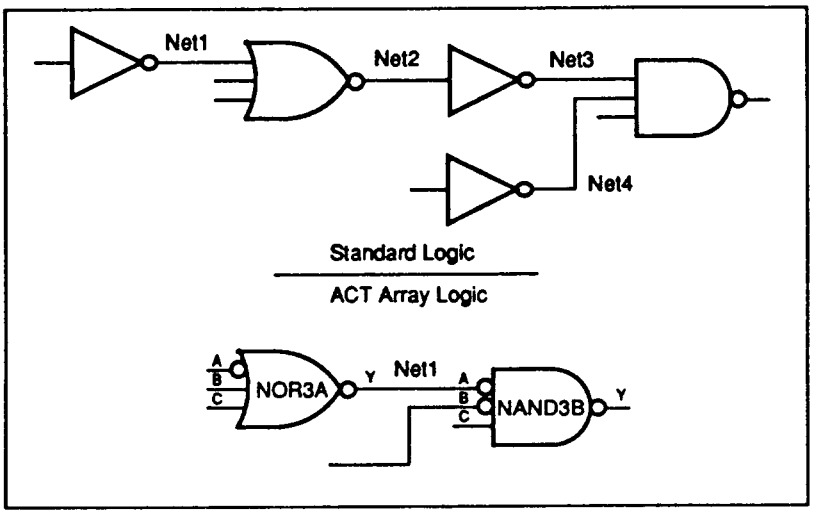


Figure 3.6.1. Reducing Logic Complexity

Act2

Before proceeding to a detailed description of design considerations it would be useful to review some fundamentals of the Act2 architecture. The Act2 architecture features two types of modules. Combinatorial modules (C-modules) are used to implement any combinatorial function in the Act2 library. Sequential modules (S-modules) can be used for either sequential functions (e.g. flip-flops) or combinatorial functions or both. When the S-module is used to implement both a sequential and a combinatorial function (e.g. a gate followed by a flip-flop) it is being used in the most efficient way. The ALS software will automatically combine a flip-flop and a preceding combinational macro into one S module where possible and perform other simple local optimizations such as eliminating unused modules in soft macros. Whenever a flip-flop and the combinatorial function driving it are combinable, and the net between them drives no other macros, then both will be combined into an S-module.

The module types exist in roughly equal numbers on Act2 devices and the place and route software will automatically select the appropriate module for each library component in the schematic.

The Act2 library is compatible with the Act1 library. Because the Act2 C-module is more powerful than Act1 modules it can implement more functions in a single

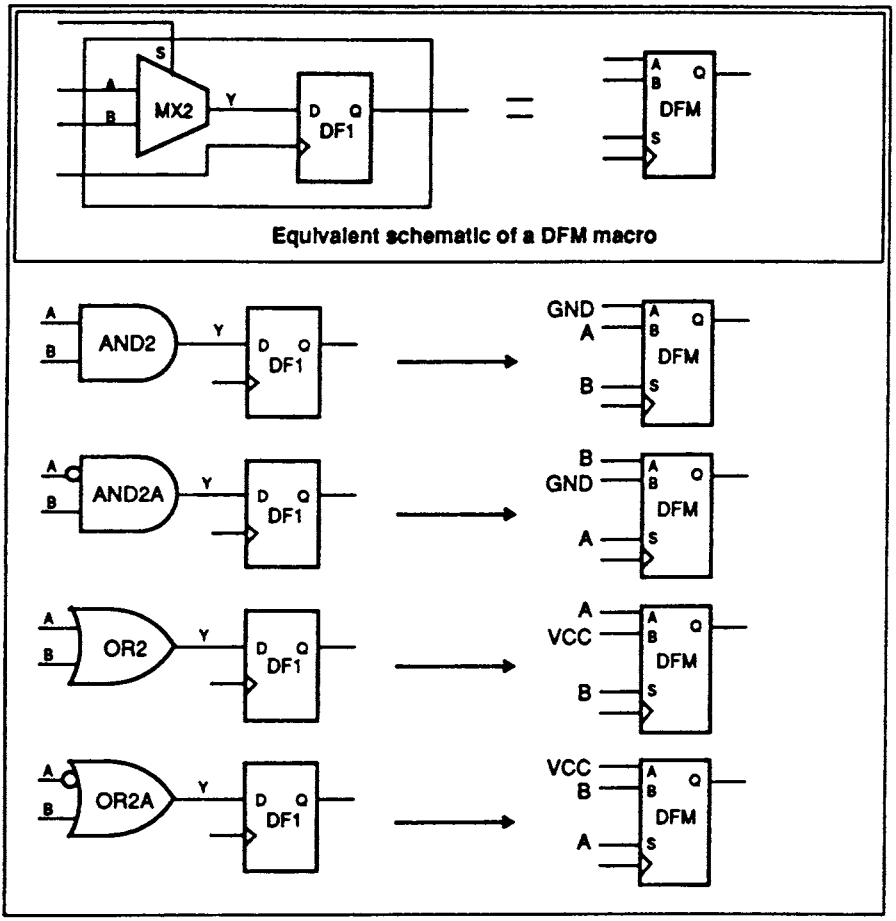


Figure 3.6.2. Compacting Logic

module. All the three-input gates and all but four of the four-input gates are single-module macros. There are also four five-input gates that require only one module.

Because about half of the modules on Act2 devices contain dedicated flip-flops, the function is inexpensive in terms of resource requirements compared to Act1. The abundance of flip-flops coupled with their high performance means that they are more attractive as a sequential element than latches.

Flip-flops may be implemented using an S-module, an S-module and a C-module, or

(as in Act1) two C-modules depending on functionality. The ALS documentation describes how each function is done and whether it is combinable.

The Act2 device has a built-in latch on each IO pad. The latches can be used as storage element for inputs, outputs, or bidirectional IOs. An IO latch may be used with an internal latch to make an IO flip-flop.

Designing with ACT FPGAs: A TTL Perspective

Actel FPGAs offer many advantages over traditional technologies such as TTL. The advantages include greater reliability and reduced board space and power from the ability of FPGAs to integrate large amounts of logic into one device. For example a single A1280 FPGA holds the equivalent of 165 MSI TTL devices (assuming 40 gates/MSI device). That means not only a smaller PCB, but a simpler one since most of the designs connections are made inside the FPGA by the 100% automatic place and route software.

Designers who are used to using TTL components may see some of the advantages of using Actel FPGAs in their designs and not realize how easy it is to begin using them. It's not necessary to know anything about the architecture of the FPGA. In fact, the schematic entry and simulation process is the same as it was with TTL. The placement and routing are done by software tools that are analogous to PCB software.

Actel provides a library with the system for popular schematic design tools. The library contains both hard macros and soft macros. Hard macros are similar to SSI components. They form the basic functional building blocks such as gates and flip-flops. Many Actel hard macros are identical in function to TTL parts though they have different names. The Actel databook contains a cross reference guide showing the names of hard macro components that match functionally to TTL.

Soft macros are more complex functions that are built from some number of hard macros. They include counters, decoders, and adders of various sizes. Some of the soft macros in the library have identical functions to MSI TTL parts. These may be identified by their name beginning with TA instead of 74. The rest of the name matches that of the TTL name. Other soft macros offer generic logic functions.

All soft macros are easily copied and modified, and there is no limit to the number of custom soft macros you may add to the library. Should you need a TTL function for which there is no near equivalent in the library it is easy to build from scratch. Simply copy the schematic as shown in the TTL databook using components such as gates and flip-flops from the Actel hard macro library.

As you gain familiarity with the Actel hard macros you will find instances where you can do a more efficient design than that found in the TTL databook. For example, if the book shows an AND gate driving an OR gate you may find a single hard macro containing both the AND and OR functions. Using such multifunction macros is very efficient because you get more and faster logic from the macro. Compare the

74AS161 counter schematic from a TTL databook with the TA161 from the Actel library in Figure 3.6.3 . The Act1 version of the function uses only 18 modules, or 3.3% of an A1020.

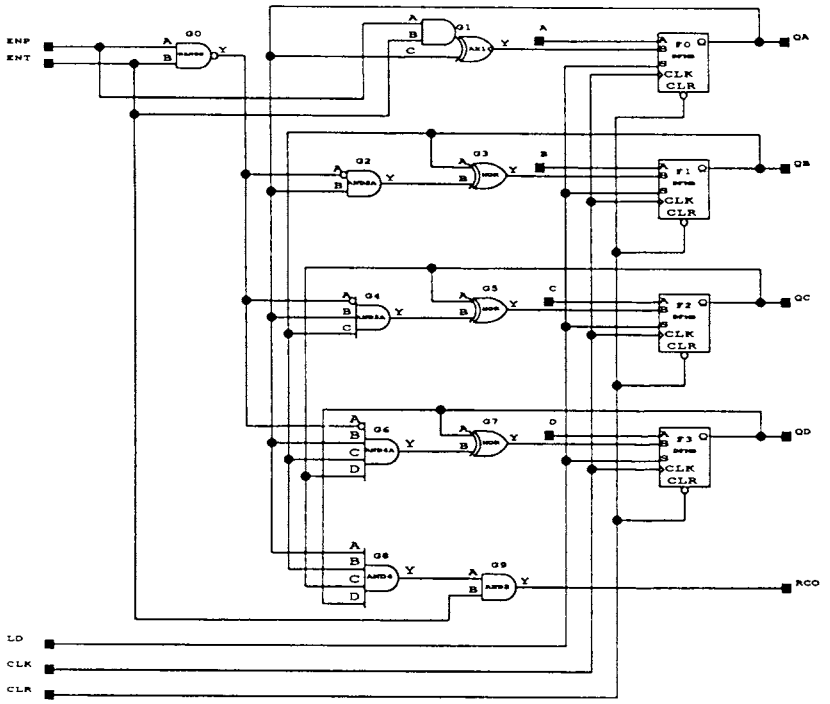


Figure 3.6.3. TA161 from the Actel Library

Many TTL parts have three-state outputs allowing them to be connected to a common bus. Three-state functions don't work well with ASICs or FPGAs because they tend to be slow and inefficient.

You don't have to give up using busses in your designs. Simply implement them using multiplexors as shown in Figure 3.6.4 . Multiplexors are very efficient on Actel parts. For example, you can create an eight-bit bus with four possible drivers using less than 3% of an A1010.

If you use a soft macro, but don't need all the outputs, you don't need to modify the macro. The Actel software contains a program called the gobbler which will

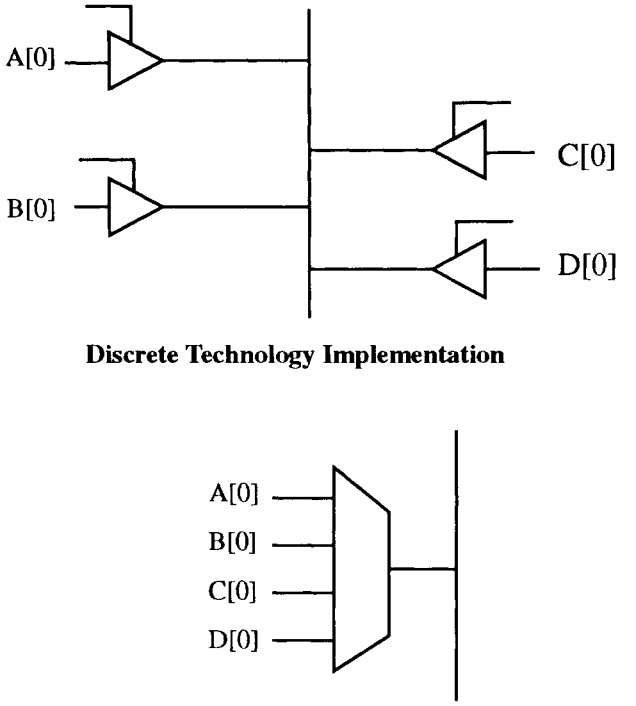


Figure 3.6.4. Least Significant Bit of a Bus with Four Possible Drivers

automatically eliminate any unused modules before the design is placed and routed.

If you use a soft macro or hard macro that has inputs you don't need the situation is different. The software won't allow inputs to be left unconnected, so some designers simply tie unused inputs to VCC or GND. That is permitted, but a better solution would be to select a hard macro that only has inputs you need or modify the soft macro to eliminate the unused function.

For example, the TA161 counter has a load function and four data inputs. Rather than tieing off those pins, a better solution would be to make a copy of the counter and modify it as shown in Figure 3.6.5 That will allow the wiring resources on the chip to be used for things other than power and ground connections.

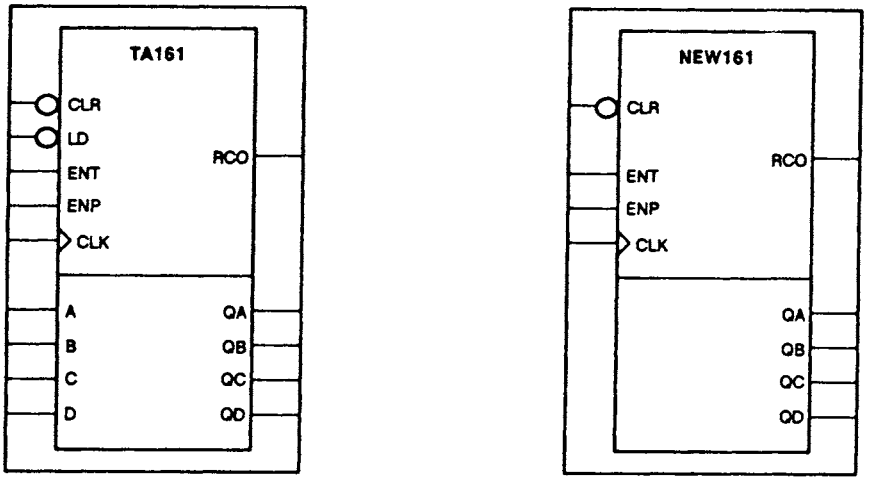


Figure 3.6.5. Old and New Versions of the TA161 Counter

Migrating PLD Designs to FPGAs

Designers who have been using discreet technology want a seamless method to convert old designs to Field Programmable Gate Arrays. They also want to have a way to enter new designs using tools and techniques they are familiar with. Tools are available that allow designers to take advantage of the logic integration capabilities of FPGAs (with the concomitant benefits of low power and high reliability) while retaining familiar design methods. The tools allow for migrations of PLD designs to be done rapidly using PLD logic description files and without drawing a schematic rendition of the design.

While the PLD migration tools can insure functional compatibility with the PLD version of the logic, designers rightly wonder about such issues as performance, efficient use of FPGA resources, and integration of the PLD logic with other functions now on the FPGA.

We'll explore some of the issues and provide a design flow for migrating PLD designs to FPGAs using the Actel synthesis tool ACTMAP which accepts a PALASM file as input and produces a netlist of Actel library components. This particular synthesis example is used for illustration only, Actel supports many different synthesis tools, and the results of each will vary, but the basic approach to synthesis is similar.

Unlike PLDs, the performance of a function implemented in an FPGA can vary.

Variations come from differences in fan-out and placement of logic modules. While it is not possible to know precisely what the performance of a function such as a state machine will be prior to placing and routing a design, pre-route estimates are reasonably accurate. Performance requirements for the PLD design on an FPGA can be stipulated to synthesis tools.

In order to take advantage of the logic integration capability of an FPGA, synthesis tools that target such devices should implement functions using the fewest number of logic modules possible. Sometimes the fastest design is also the smallest in terms of number of modules used. In other cases, the synthesis software can introduce parallel logic to reduce the number of logic levels and improve performance.

One of the keys to good synthesis software, and one of the most difficult to achieve optimally is the efficient use of logic modules in synthesizing logic. Tools must be capable of searching a library of cells and select those that provide the highest number of gates per logic module. The ability to conduct such a search depends on the algorithms the synthesizer software employs.

In a discreet PLD, all the IOs connect to other devices via PCB traces. When the PLD logic is incorporated into an FPGA as part of a larger design, most of the IOs are connections internal to the device. Typically the PLD design is a state machine or counter whose outputs drive controls of data path logic such as multiplexors or registers which are also implemented in the FPGA.

Actel devices use multiplexors rather than three-states when a design requires multiple drivers for a line. There is a slight difference in migrating PLD designs to an FPGA depending on whether the three-state in the PLD was used or not. If the PLD three-state control is a pin and was not used in the original design, simply changing the pin's name in the pin list to NC will signal the Actel synthesis tool to ignore the unused PLD three-state functionality. The synthesized logic outputs can drive other logic in the FPGA. If the design requires that the synthesized logic outputs drive a three- state IO pad, simply assign a signal name to the three-state pin position in the pin list.

For PLDs with internal three-state control, such as the 22V10, the situation is analogous to a three-state control pin. The logic controlling the three-state pin on the PLD can be synthesized to drive a three-state IO pad on the FPGA, or to be an internal output to control a multiplexor select line.

Multiple PLD outputs can be used to drive a line if the internal three-state control lines are activated by mutually exclusive logic functions. An example of such an application may be seen in Figure 3.6.6 where an equation defines the three-state output enable logic. Like the other logic in the PLD, the three-state output enable control is entered in a sum-of-products format and implemented in the PLD AND-OR array.

When migrating a discreet logic design which includes PLDs to an FPGA all the logic

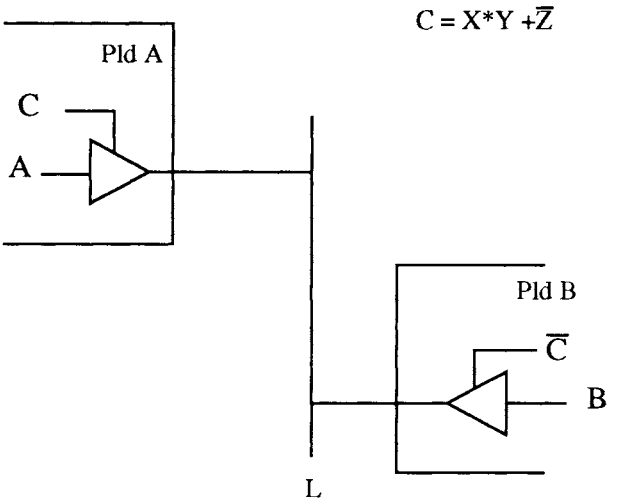


Figure 3.6.6. PLD Three State Output Enable Logic

is typically in a single device. Internal three-state functions may not be available, but even if available they tend to be slow.

An alternative to the internal three-state would be to use a multiplexor as shown in Figure 3.6.7 . A logic synthesizer can replace the three-states with the multiplexor and synthesize the output enable control logic to drive the multiplexor select line.

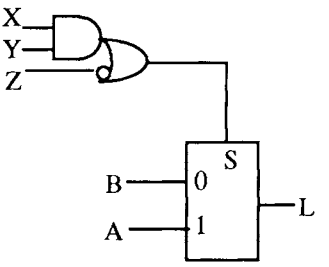


Figure 3.6.7. FPGA Multiplexor Alternative

Synthesis Design Flow

The design flow for the Actel Synthesis is shown in Figure 3.6.8 The designer may

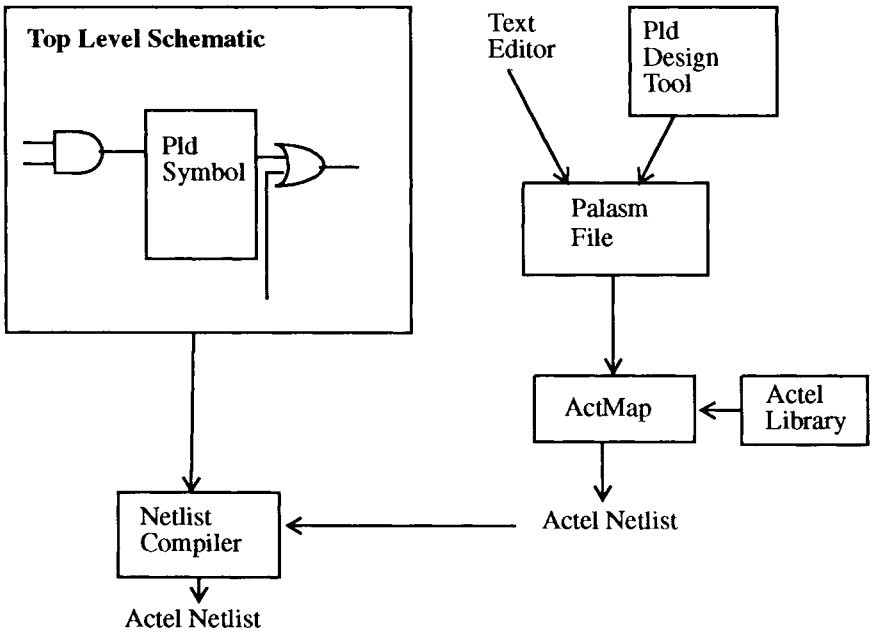


Figure 3.6.8. ACTMAP Design Flow

use any design entry tool that can output a PALASM-format file. Popular tools that can output the format include ABEL, CUPL, PLDesigner, and LOG/iC. Alternatively, a PALASM file may be written using a text editor.

After taking the input file, the synthesizer will generate a suboptimal netlist and proceed to optimize it. The user may influence the optimization process by using command line arguments to optimize for minimal capacity or maximum performance. The latter argument stipulates a performance goal that the tool tries to meet.

Since the actual delays for the complete design aren't known, the synthesizer assumes a typical routed delay value. If it cannot meet the criteria, it will report the performance it did achieve. The resulting netlist is in the Actel ADL format and resides in a separate directory.

The designer must create a symbol to be instantiated in the schematic that represents

the synthesized netlist. The symbol IOs correspond to those defined in the PALASM file. It is marked with a property indicating that the netlist was synthesized and is to be found in a separate directory. When the top-level design is compiled, the schematic and the synthesized netlists will be integrated into a single file.

The ACTMAP tool also generates a netlist in the format of the users CAE tools which may be used to simulate and verify the synthesized logic either alone or interacting with other logic in the design. Schematic generators may be used to view the synthesis results.

Designing Counters with ACT Devices

Perhaps the most common digital logic function used is the synchronous binary counter. Regardless of the technology employed to implement counters, they are found in every type of application. The following describes some of the techniques for designing counters, or modifying a counters in the Act1 and Act2 soft macro libraries.

The functional requirements of the counter determine the approach to be taken in its design. Most applications minimally require synchronous counters with an asynchronous reset.

A simple binary counter with reset is shown in Figure 3.6.9 . The least significant bit uses an inverting flip-flop so it toggles without additional logic. Whenever counter output polarity is flexible, or parallel flip-flops are warranted by fan-out or performance considerations, it allows the design to take maximal advantage of macro bubbled IOs to reduce module count and levels of logic.

In the simple counter, as in most types of counters, the biggest design concern is propagating flip-flop outputs to higher bits. For Act1-based designs like the simple counter example, bits higher than eight require another level of combinatorial logic. Combinatorial levels required for the simple counter are: two for up to nine bits, three for ten to 33 bits.

A loadable counter appears in Figure 3.6.10 . The multiplexor flip-flip is used where one input provides the loaded data and the other the count data. The multiplexor select line is controlled by the counter load line.

The Act2 modules offer more logic integration than Act1 modules. In the sample design of Figure 3.6.11 you can see how to construct a six-bit loadable counter with only one level of combinatorial logic between flip-flops. (In the counter design described below a ten-bit counter with one level of combinatorial logic may be built using a multiplexed flip-flop with a gated select input.) The design has a 4:1 multiplexor driving the data input of each flip-flop. The multiplexor macro and the flip-flop are combinable into a single S-module by the ALS software providing good integration and performance.

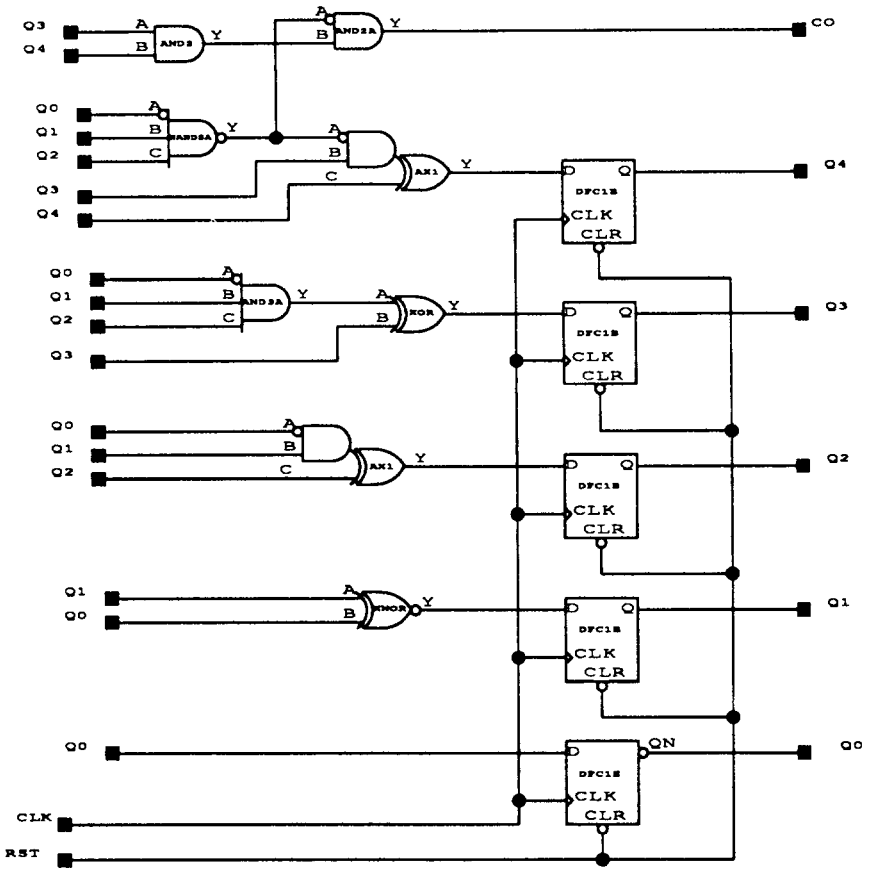


Figure 3.6.9. Act1 Five Bit Binary Counter

The select lines on the multiplexor are operated by the load control (S1) and by the count enable and carry from the lower order bits (S0). The multiplexor data inputs are used for data to be loaded, held, or incremented. The fourth multiplexor data input can be used as a second data input, or a synchronous set or clear.

Large counter design considerations should be evaluated from the perspective of the device-level design and how the counter is used in it. For example, if some counter outputs may be active low or if additional modules are used for redundant logic (e.g. some bits have both an active-high and an active-low output), then larger counters may be designed without additional logic levels using five-input gates. Such decisions should consider the implications for module count and fan-out as detailed below.

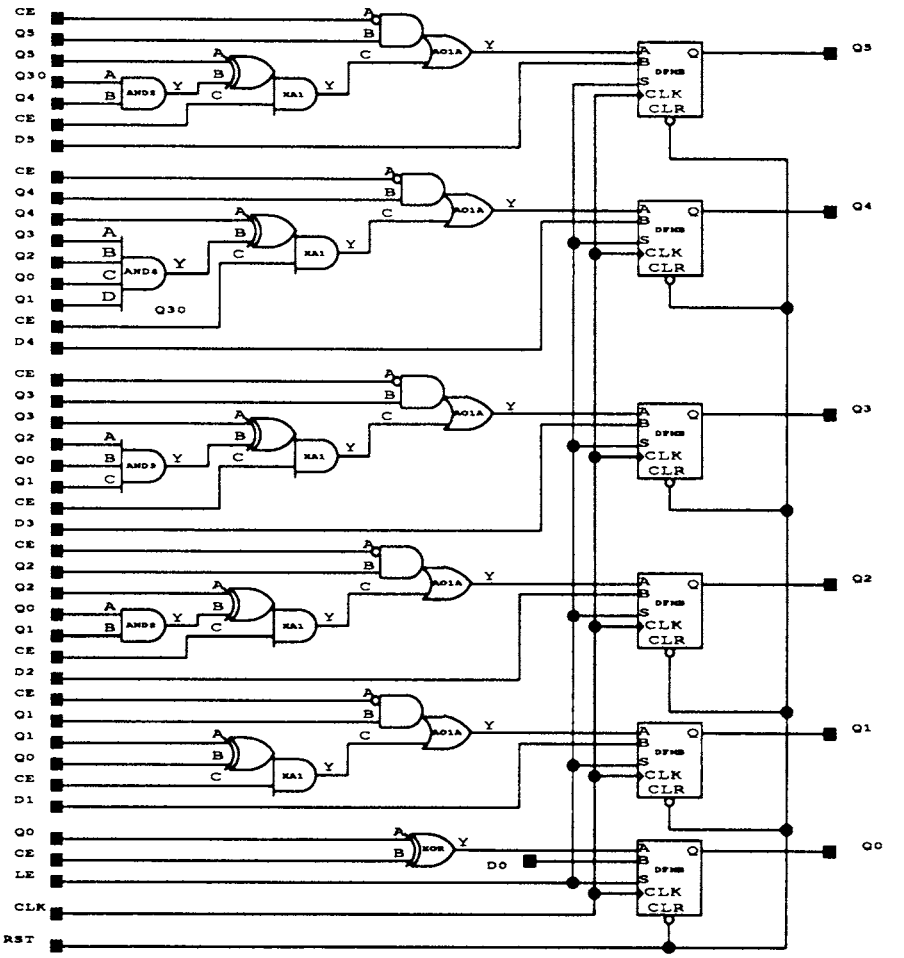


Figure 3.6.10. Act1 Loadable Counter

The sample design for a 16-bit synchronous loadable binary counter with a count enable will serve to illustrate some of the considerations designers should be aware of when designing large counters in Act2 FPGA designs. The functional description for

the counter appears in Table 8

Table 8: Counter Function

Reset	Load	Counter Enable	Clock	Q
0	X	X	X	0
1	1	X	Rising	D
1	0	1	X	Q
1	0	0	Rising	Q+1

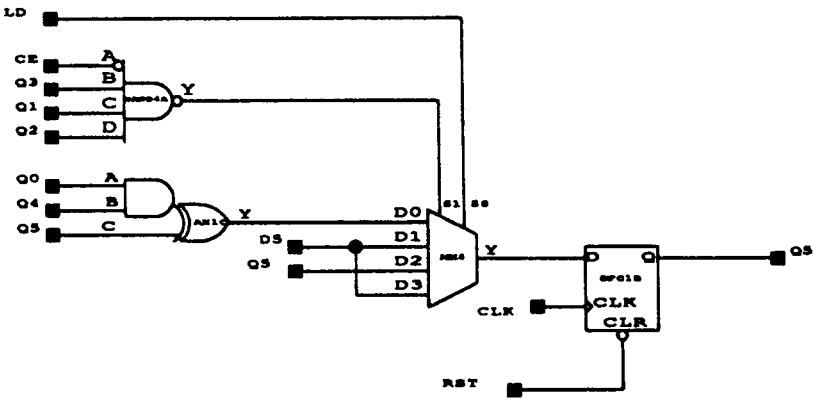


Figure 3.6.11. Act2 Six-Bit Loadable Counter Example

The counter design makes extensive use (bits 0 through 5) of a 4:1 multiplexor driving a flip-flop as depicted in Figure 3.6.12 . C-modules are used as AND-EXORs and for ANDs to qualify the count function. The AND function is also used to bring the count enable (*CE*) to the multiplexor via the select line in bits Q3 and above. Using both the select inputs as well as the data inputs to AND lower order bits allows for more paralleling resulting in fewer levels of logic.

For the bits Q6 through Q15, an S-module macro with both a 4:1 multiplexor and an OR gate driving one select line is used to allow for more parallel propagation of the lower bits. Figure 3.6.13 shows the implementation of the most significant bit of the

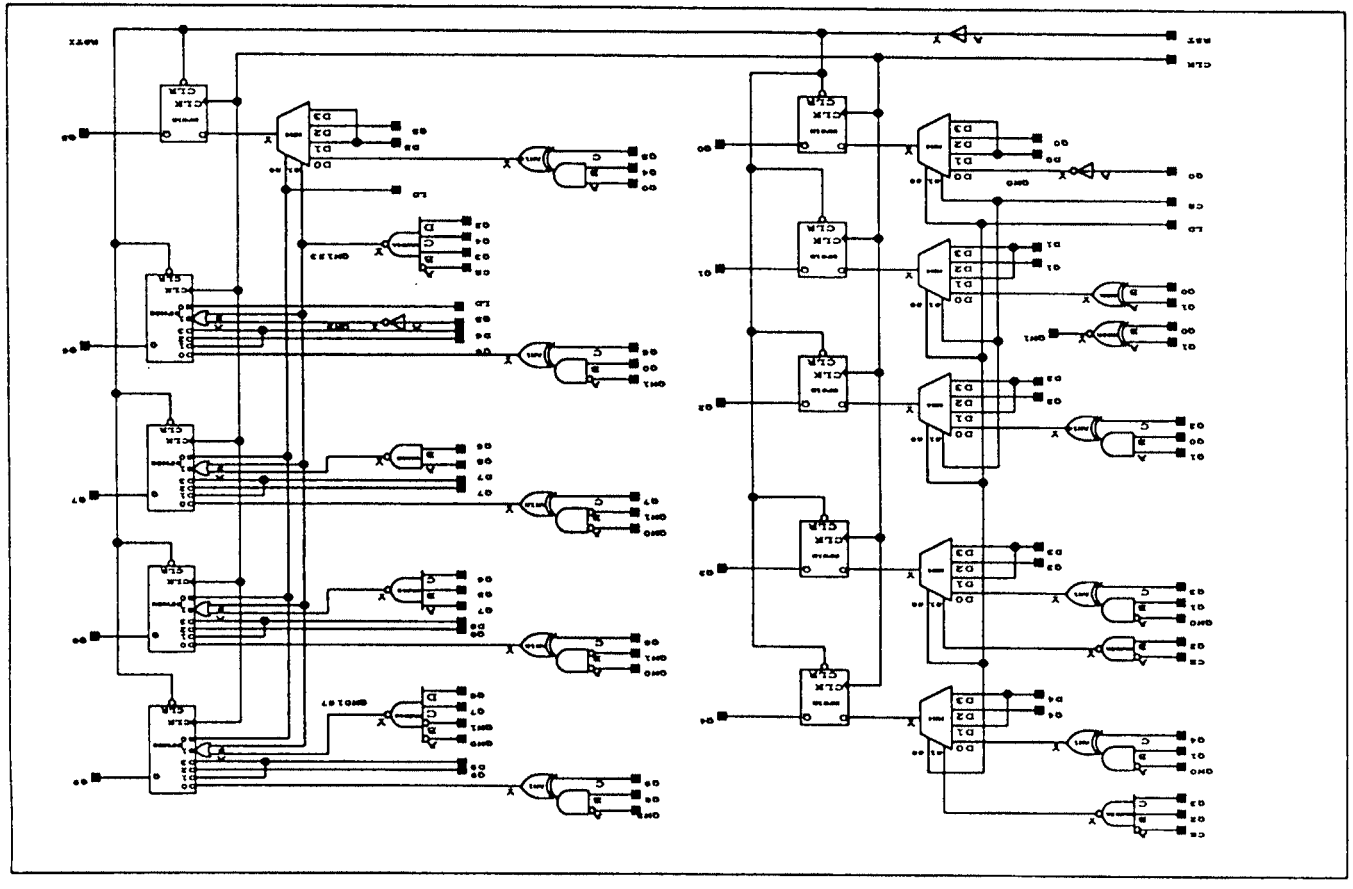


Figure 3.6.12. Counter Example Using Multiplexor And Flip-Flops

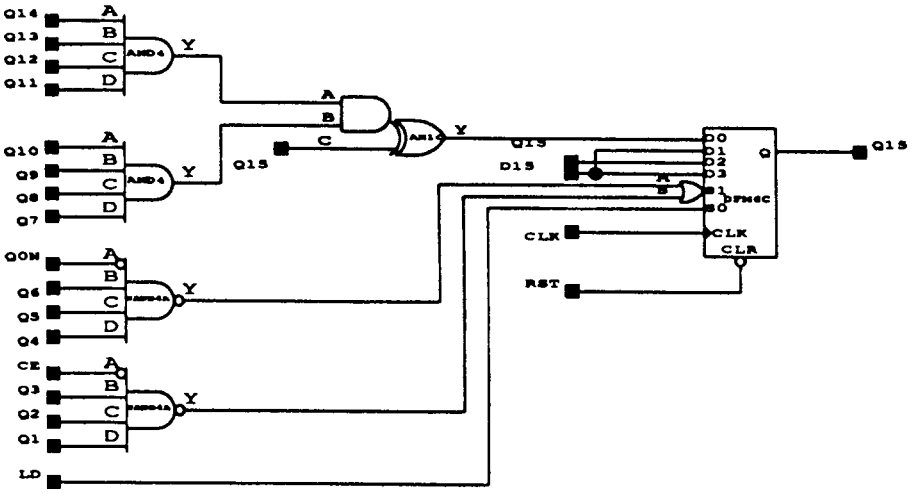


Figure 3.6.13. Counter Most Significant Bit

counter.

The S-module OR gate is used as a two-input NAND with active-low inputs which are, in turn, driven by NAND gates to propagate the lower bits and the count enable. The active-low output of the built-in two-input gate (OR used as a NAND) is adjusted for by shifting the position of the multiplexor data inputs.

Most four-input gates are implemented with a single C-module, but a four-input NAND with no bubbled inputs requires two modules. The limitation is avoided by using a NAND with a bubbled input. The count enable is active low, so it may be used to drive a bubbled input on a gate. Active-low counter bits are used to drive other bubbled gate inputs.

As may be seen in the schematic of Figure 3.6.12 , two bits (Q0 and Q6) use inverters. The Q0 inversion toggles the flip-flop. A toggle flip-flop could have been used instead of a D flip-flop, but it could not have been combined with the multiplexor into a single S-module. Moreover, the inverter output is available as a resource to share the fan-out load with the flip-flop and to allow the use of bubbled inputs on gates whenever it is desirable.

It could be argued that the use of the inverted output to drive gates causes the lower level bits to use two levels of combinatorial logic when it is not necessary. For a design of ten bits or less the point would be valid because no path requires more than one combinatorial level. In the example design, however, two levels are already required by the upper bits and the improvement in fan-out from the use of the inverter output at no additional cost in module count makes the practice worthwhile.

The paths in the design that are most likely to limit performance are those with the largest number of logic levels and the highest fan-out. In general, when fan-outs exceed nine on a critical path, using redundant logic is often clearly called for. For lower fan-outs the decision to use redundant logic is problematical and must be balanced by considering both the cost in additional logic modules as well as the fan-out to the outputs driving the redundant logic.

In the sample design, two redundant modules were added to illustrate the concept. One is an XNOR gate whose output is the inversion of Q1. The other is a four-input NAND gate which propagates flip-flop outputs. No fan-out in the design exceeds seven and the worst-case path is the redundant gate whose inputs are driven pins with fan-outs of seven, six, six, and five; and whose output fan-out is four. A total of 48 modules were used in the design.

Pre-scaled Counter Design

An important realization in designing high-performance counters is the fact that the least significant bits of the counter change the most frequently, and higher order bits change much less often. This fact can be used to optimize counter performance by making sure the least significant bits propagate up at the fastest rate. Higher order bits have a longer time to propagate through the logic. Lets consider the design of a loadable six-bit counter using the technique. It will be a down counter, suitable for timing or address generation.

The counter needs to have a least significant bit which can toggle at the highest possible rate. In the Act2 family the sequential logic module allows a 4-input multiplexer with gated select lines and a D-type flip-flop to be implemented in a single level of logic. The logic module can be used to construct a least significant counter bit with clear, load and count enable as shown in the upper portion of Figure 3.6.14 .

Data can be loaded into the register when the load enable (LD) signal is high. The count enable signal (CNT) is used to toggle Q0 when it is active and hold Q0 when inactive. The circuit implements a least significant bit which needs only a single level of logic to operate.

Figure 3.6.14 shows the implementation of Q1 using the Act2 module. It is similar to the LSB except Q1 is inverted to toggle the flip-flop. Since Q1 can only toggle when Q0 is a zero (in a down counter), we will have an extra clock cycle for the inversion to propagate for Q1's next state. That allows the slower /Q1 time to settle, insuring fully synchronous operation. The basis of the design technique will be to have slower signals gated off until they have had sufficient time to settle and they are needed to compute the toggle of the associated counter bit.

Added to Q0 and Q1 are three registers to source CNT, /CNT and LD as shown in the lower part of Figure 3.6.14 .The complete schematic illustrated Figure 3.6.14 show a

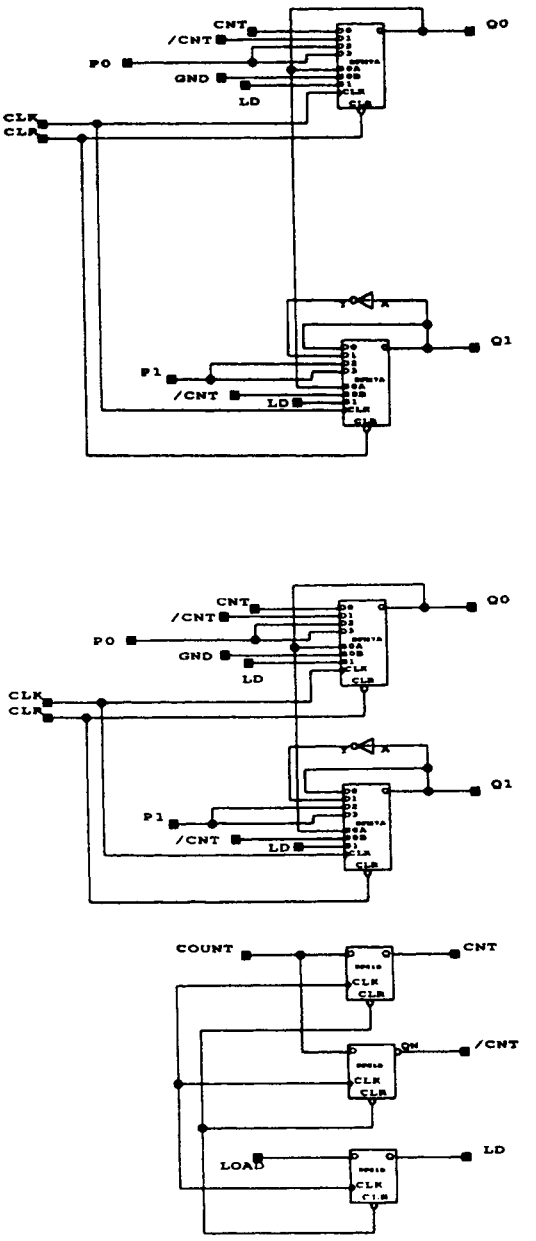


Figure 3.6.14. Act2 Pre-scaled Counter

2-bit pre-scaler (later referred to as CNT2P) for fast counters.

A four-bit macro for the MSBs of the six-bit counter is shown in Figure 3.6.15 . It

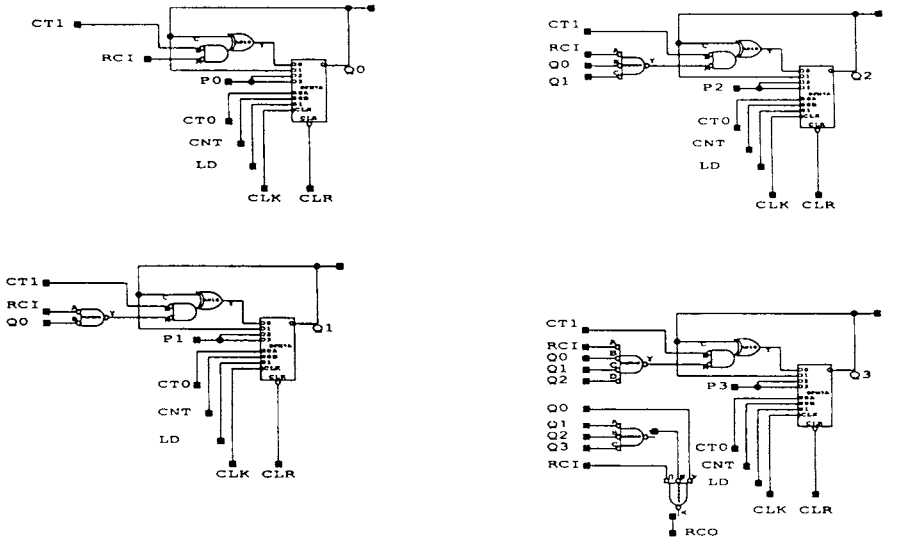


Figure 3.6.15. Four Bit Counter Macro

uses the LSB from the CNT2P macro (connected to CT0) and CNT to enable the multiplexer inputs used for toggling. The next LSB (Q1) from CNT2P is connected to CT1 along with the counter bits in the macro (Q0-Q2) to gate the input to the XOR determining the next count. A ripple carry input (RCI) is also used to allow results from previous stages to participate. This technique allows the MSBs four clock cycles (the time for the LSBs to transition from 0 through 3) to develop the input to the associated bits in the counter. The complete six-bit counter is shown in Figure 3.6.16

By replicating the pre-scaler, fan-out limited counters can be constructed. As an example, an 18-bit counter implemented with the above macros is shown in Figure 3.6.17 . The limiting frequency for the counter is determined similarly to the six-bit

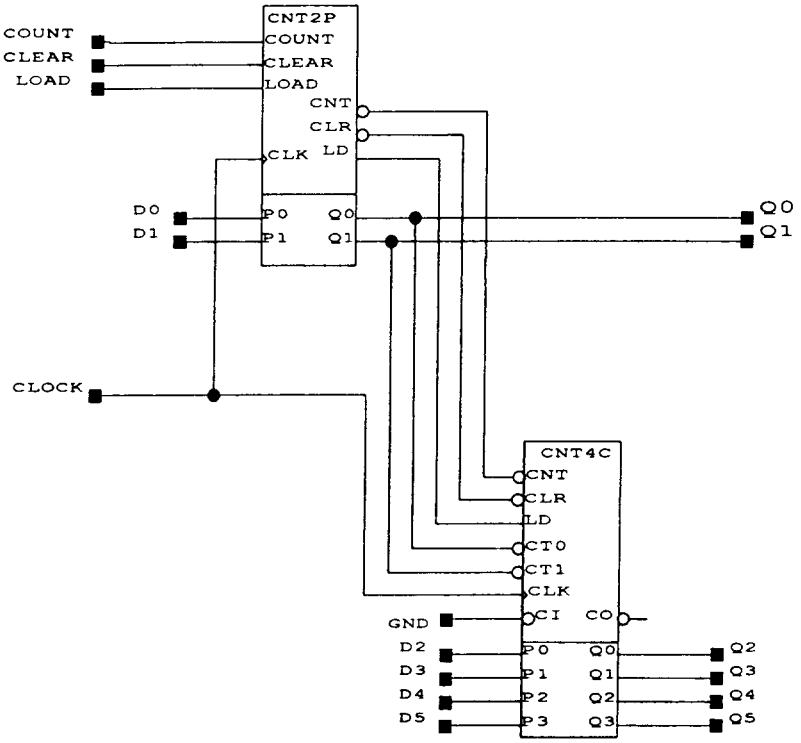


Figure 3.6.16. Six Bit Counter

example.

Designing Adders and Accumulators with the ACT Architecture

Many designers implement adders using carry-propagation techniques. The multiplexed-based Act1 and Act2 combinatorial module (C-module) allows for the more efficient carry-select design. This method partitions the add function into blocks that perform two additions simultaneously on a number of bits of the two operands.

The two additions are performed simultaneously except that one assumes a carry-in and one has no carry-in. The two sums are input to a 2:1 multiplexors, one for each bit pair. The carry line, from the low bits to the high bits, is used to select the appropriate sum for each block.

The ACT architecture lends itself well to implementing adders of various sizes using

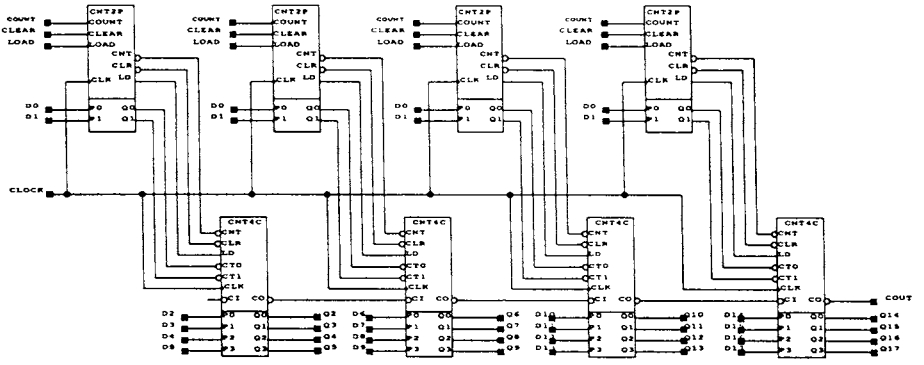


Figure 3.6.17. 18-Bit Counter, Fanout Reduced Design

the carry-select technique. A sample design for a 16-bit adder, as shown in Figure 3.6.18 , will be used to illustrate adder design.

The method for obtaining optimal performance from a carry select adder is to design it such that the number of levels of logic required for the carry chain equals the number for the largest sum block as closely as possible. When they are the same number of levels, the sum bits arrive at the data pins and the carry arrives at the select pins of the output multiplexing stage simultaneously.

The way to balance the levels of logic modules for the sum blocks with the carry is to partition the sum blocks. This partition is based on the logic levels required for the sums and the levels for the carry between sums. The size of the partitions varies with width of the data. The Act2 library contains some powerful hard macros that are used to shorten the levels of logic required for generating sums and carries. The description of the sample design will illustrate the use of the macros.

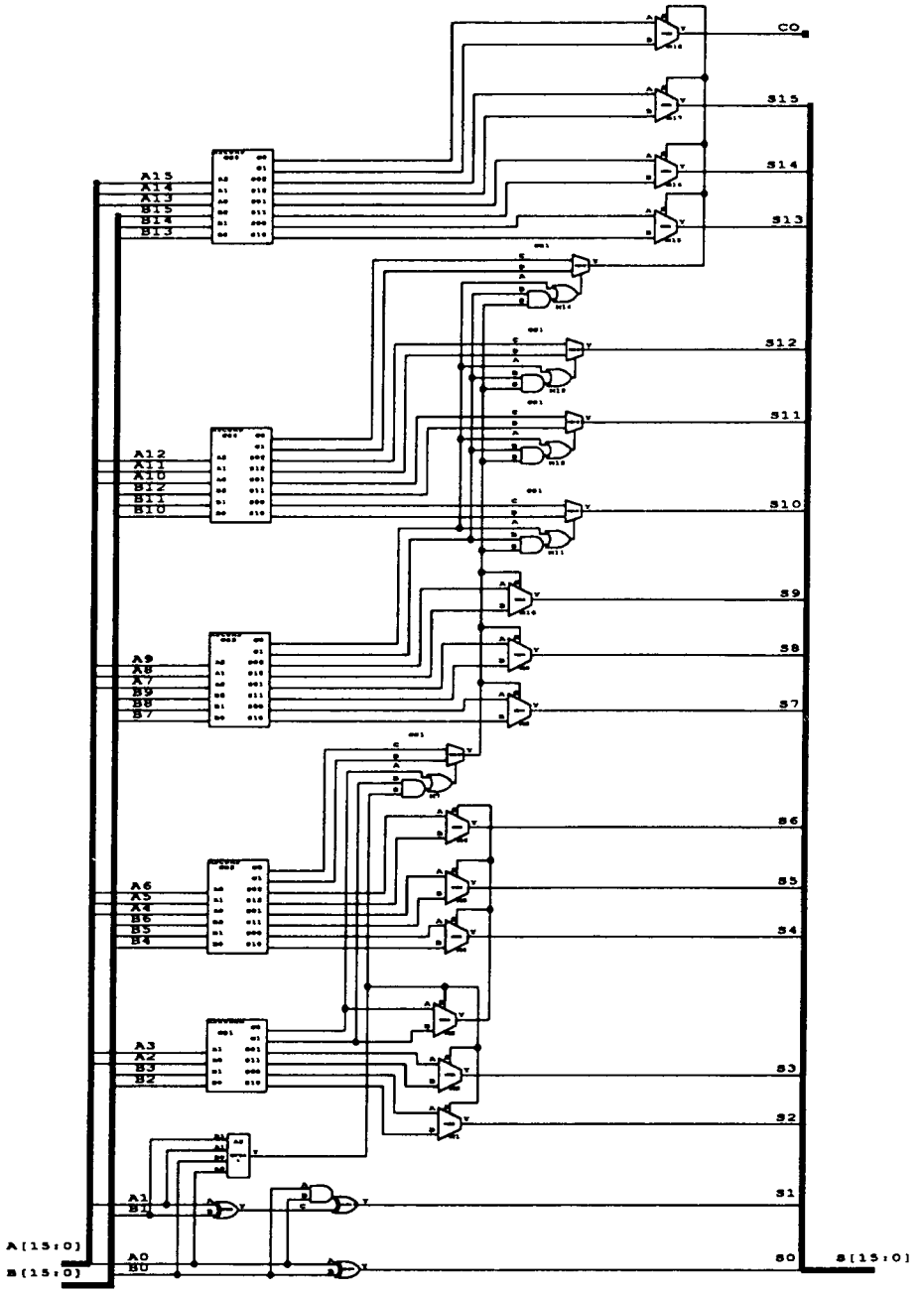


Figure 3.6.18. Act2 Carry-Select Adder Example

For the 16-bit adder, the optimal organization is to perform two two-bit additions on the least four significant bits with the remaining higher order bits broken into four sections of three bits each. In the top-level schematic the addition logic of the two least significant bits is visible. The other additions are performed in lower levels of the design hierarchy described in the next section.

The Act2 library includes two two-level carry hard macros. One macro generates a carry for the two bit pairs assuming the carry-in is true and the other assumes it is false. The latter macro may be seen at the bottom of Figure 3.6.18 making the carry for the least two bits.

The carry macro output drives the select line for the 2:1 multiplexors for sum bits two and three. It also drives the select line on the cascade multiplexor. The cascade multiplexor is a special Act2 hard macro that can propagate two levels of carry. The macro is depicted in Figure 3.6.19 and has five inputs. The top multiplexor inputs

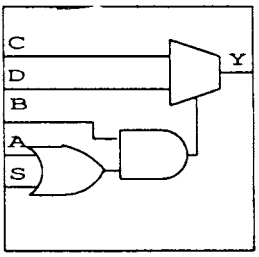


Figure 3.6.19. Cascade Multiplexor Macro

select the most significant sum or carry. The lower three inputs drive logic that implements a simplified form of a 2:1 multiplexor.

A fully implemented 2:1 cascade multiplexor does not map into the Act2 module efficiently, but the full functionality is not required in a carry select adder. A simplified version of the cascade multiplexor that maps into a C-module or that can be combined with a flip-flop in an S-module is available.

The simple version has logic driving the select for the upper level multiplexor consisting of only a two-input OR driving one input of a two-input AND. The two OR gate inputs are driven by the carry output from the next lower sum block assuming no

carry-in and the carry in from the rest of the lower bits of the adder. The remaining AND input is the carry from the sum block which assumes a carry-in.

The logic is correct for a carry select adder because if the assume-no-carry-in input is true (meaning that a carry was generated within that sum block), then the assume-carry-in is always true (since it equals the false plus one) which completes the AND function.

If the carry from the lower bits is true (meaning a carry is propagated to the sum block), then we complete the AND if the assume-carry is true.

The schematic for the three-bit adder block appears in Figure 3.6.20 The adder

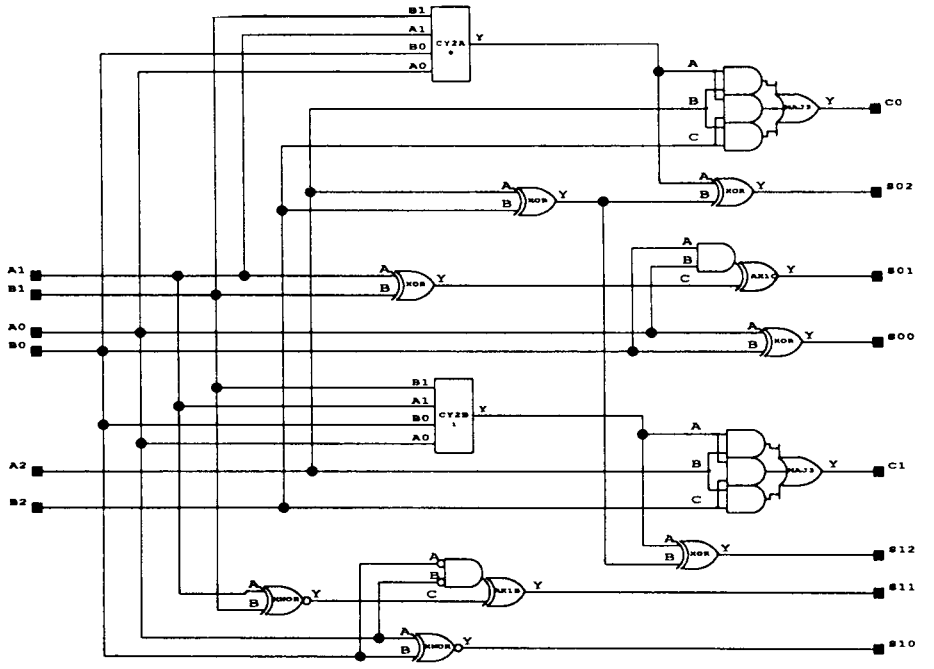


Figure 3.6.20. Three Bit Adder Block

requires thirteen logic modules to generate the three sum and carry pairs. All the output paths are two levels of logic or less. The two carries for the three bits come from two-level carry hard macros driving a three-bit majority macro. All the sums are generated from exclusive OR or NOR gates.

The optimal design for a carry select adder depends on the number bits to be added. As mentioned previously, the number of bits in a block is a function of the overall adder size. For Act2 adders of two and three bits, the design shown in Figure 3.6.20 is attractive because the two-level carry macro allows for delays to be two levels or less.

An example for another design for an adder block is shown in Figure 3.6.21 . This

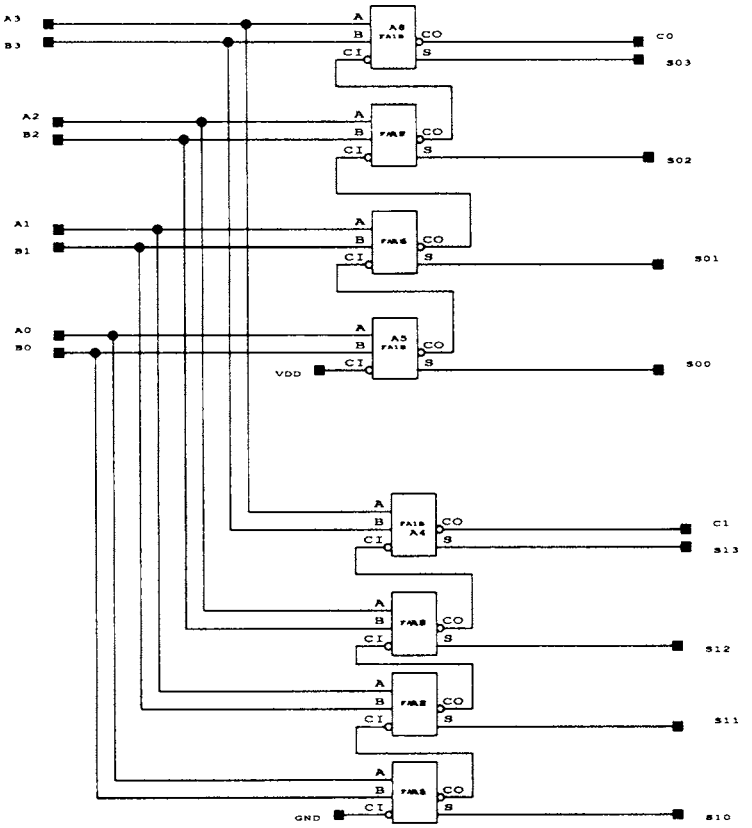


Figure 3.6.21. Another Adder Example

design uses ACT library single bit two-module hard macros. The Actel library contains several such two-module adder macros with both a sum and a carry output. The carry output goes through one module delay and the sum output two module delays. The carry propagates through the chain in four levels in the example so that all the sum bits are stable when the carry ripple is complete.

A three bit adder block has one level of delay more than that of Figure 3.6.20 , but requires one less module. For Act1 adder blocks and Act2 blocks greater than three bits, the carry chain design is most efficient.

Accumulator

All the sum and carry outputs of the adder macro are combinable into a single Act2 sequential module (S-module). This combinability feature means that if the data inputs of a 16-bit register are schematically connected to an adder's outputs, the ALS software will automatically put the adder output macros (2:1 multiplexors or cascade multiplexors) into their respective flip-flop in the register.

The registered-output adder will suffer no degradation in performance from the combining because the delay through the combinatorial part of the S-module is less than that of an uncombined macro. Tieing the register output back into the inputs will make the circuit into an accumulator. A sample design for an accumulator made from an adder and a register may be seen in Figure 3.6.22 .

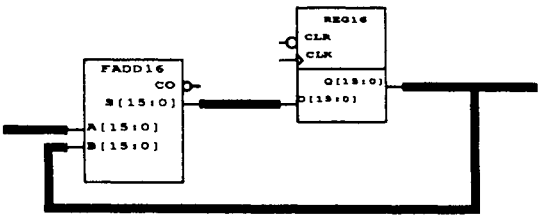


Figure 3.6.22. 16 Bit Accumulator

Design Results

The sample design uses 82 Act2 modules for either the adder or the accumulator. The slowest path in a function is usually the one with the most levels of logic. In this case it is the carry chain which has four levels of logic. As mentioned previously, all other paths have fan-outs of three or less.

The modules in the chain have fan-outs of three, four, seven, and four. Criticality may be used to optimize the path performance. Criticality works best when fan-out is low. When the fan-out of a speed-sensitive net exceeds seven, performance can usually be most improved by adding redundant logic. For fan-outs of less than seven, adding

redundant modules may not bring any improvement. Using redundant logic for fan-outs of seven should be considered on an individual basis. Adding a redundant module to the carry path would change its fan-outs to three, five, three, and four. The expense of one module may be justified by the performance improvement from lowering the fan-out.

It is also possible to improve performance by pipelining an adder. Since all of the combinatorial functions used in the adder are combinable (if the function's output drives a flip-flop ALS will put both in a single S-module), designers may pipe the adder at the points that provide the best performance at no cost in additional modules.

The carry select architecture is extensible to adders of any size. Adders of eight to fifteen bits may be designed using the technique in three levels of logic. Adders from 16 to 24 bits can be done in four levels. When adapting the adder design to other operand sizes remember to re-partition the sum block sizes to match the logic levels of sums and carries.

State Machine Design

The traditional methodology for designing state machines has been to draw a state diagram, map the states into the minimum number of register bits needed to encode them and determine the minimal next state function. The method results in a minimum number of registers, but often requires wide gating to determine the next state bit because all the register outputs are feedback to each transition term.

PLDs are register lean but can do wide gating, so their architecture fits into the above methodology easily. FPGAs however offer designers a different set of resources. The style of implementation of the state machine output coding should consider how best to take advantage of the resources.

On Act1 devices flip-flops use two modules and are relatively expensive compared to gates. For these devices, encoded outputs are often most efficient. On Act2 devices, registers are abundant and gating is optimized for more narrow functions. A state machine designed to take advantage of the large number of registers available on an Act2 FPGA may be more efficient than the PLD implementation. A sample state machine illustrated in Figure 3.6.23 will be used to contrast the encoded state method with the Bit-Per-State (BPS) technique which uses a single register for each state. We'll use the encoded method to implement the design using the Act1 library and repeat the process using BPS.

The example contains six states, seven inputs, and five outputs. The traditional approach would encode the states into three state bits by making state assignments. The schematic in Figure 3.6.24 shows one possible implementation. The logic required to implement the state machine consumes 29 Act1 modules. The longest path is through four levels of delay.

With register rich ACT devices BPS becomes more attractive. State assignment is

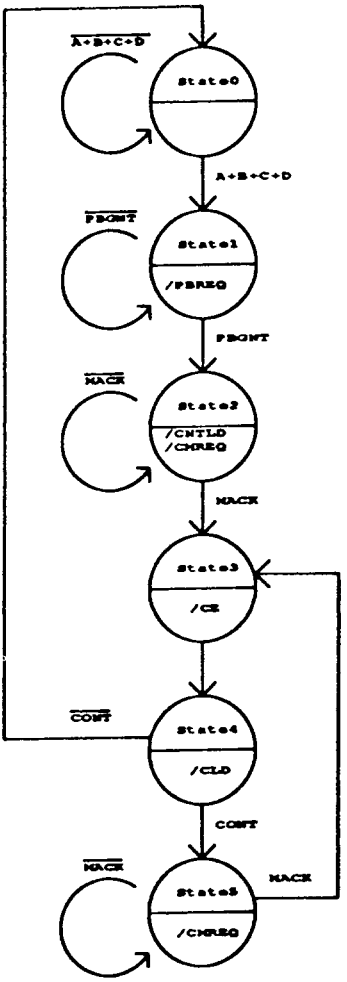


Figure 3.6.23. Example State Diagram from 4-Channel DMA Controller

trivial if a register is used for each state bit and the design will directly implement the transition terms. The schematic of Figure 3.6.25 exemplifies the results of the BPS method. It uses 19 Act1 modules and only requires two levels of delay.

Note that the state bits are inverted to make the states active low and the single

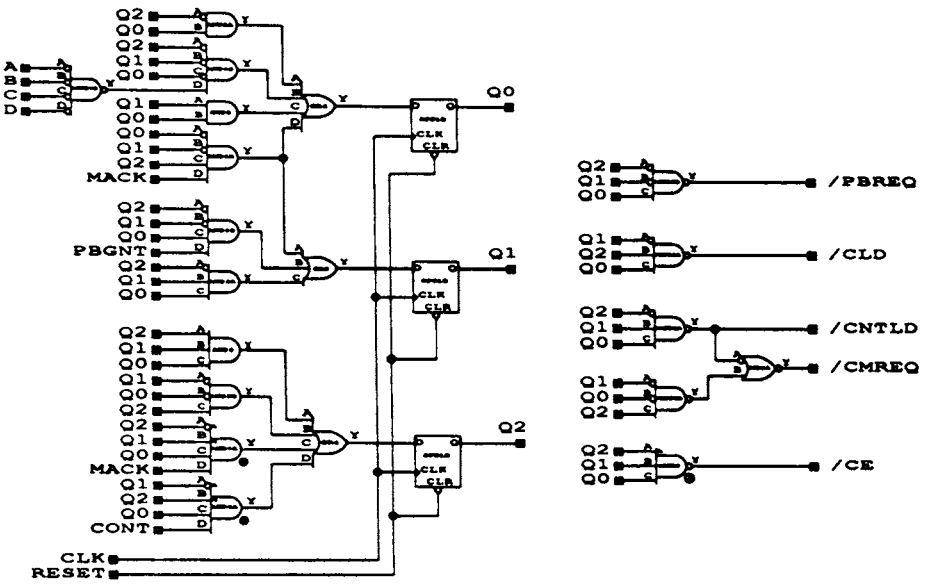


Figure 3.6.24. Act1 State Machine Schematic

module OA2A and OA2 are used to create the transition terms. The reset signal sets all states except State0 which is preset since it is the initial state for the machine. If active high state bits are required they can be used by clearing them on initialization and detecting the active high state for transition purposes.

Larger state machines may also be implemented using this technique by distributing control to several smaller machines and using a single master machine to coordinate activities. This usually results in a higher performance design since control signals will be located near the logic they affect, minimizing routing delays. In addition, typically it is easier to design and debug since each machine can be more easily understood and interactions between operations are minimized.

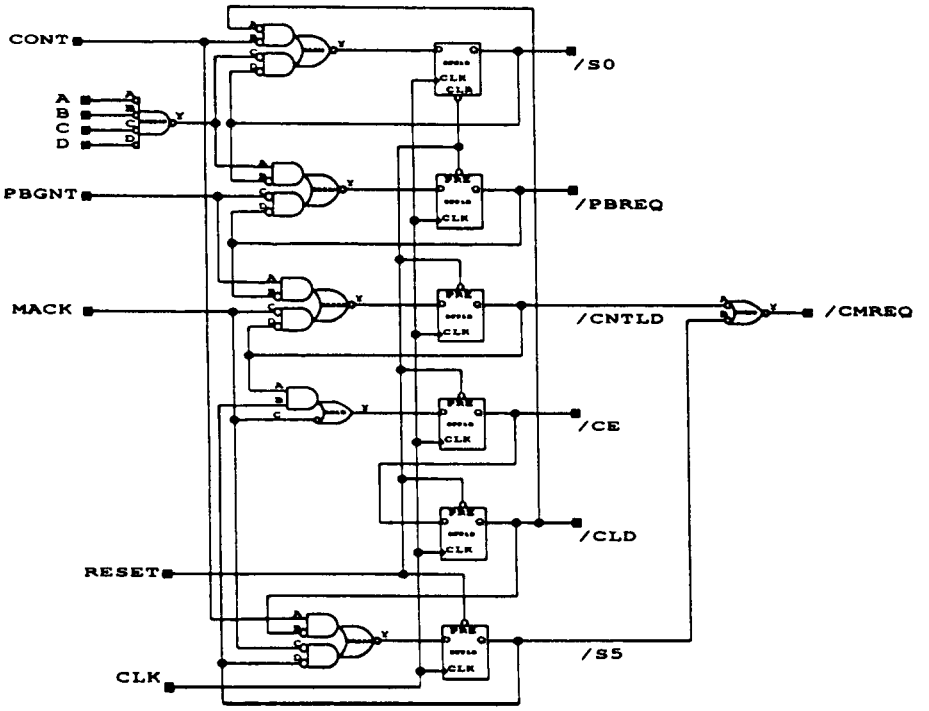


Figure 3.6.25. Act2 State Machine Implementation

A summary of the methodology is as follows:

1. Write state transition equations for each state.
2. Assign each state to a separate register. Where possible state bits should be made active low to make it easier to construct transition terms in a single ACT 1 module.
3. Take output signals directly from state outputs where possible. If the output signal must be active high, the state bit associated with it can be made to correspond by inverting its driving logic, and inverting its inputs to any other transition term.

4. All active high state bits are reset on machine initialization and active low states are preset on initialization. The initial state is must be activated on initialization so it should be cleared if active low or preset if active high.

Using FPGAs for Digital PLLs

In addition to purely digital applications, many designs use FPGAs for Digital Signal Processing (DSPs). We'll examine one such application, digital PLLs, to show two ways of implementing PLL designs using FPGAs.

Pulse Steal PLL

In telecommunications applications it is often desirable to generate a digital signal which is locked to an incoming signal and is some multiple of its frequency. A drawing of a pulse steal PLL which is a simple way generate such a signal may be seen in Figure 3.6.26 . Note that the design contains an ordinary oscillator, but no VCO. Except for the crystal, the entire design will operate in an FPGA.

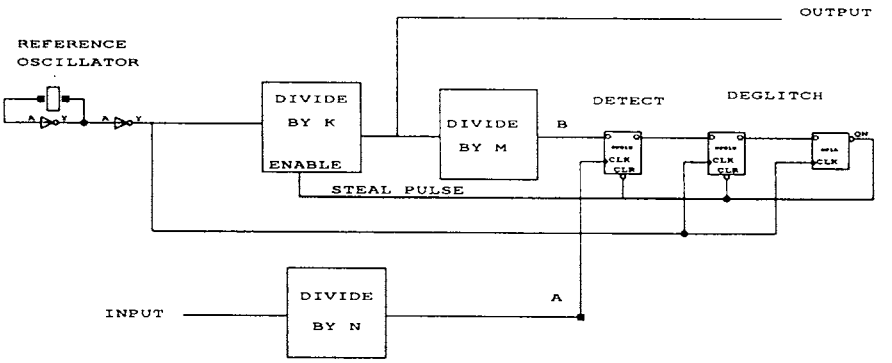


Figure 3.6.26. Pulse Steal PLL

Note the frequency relationship that holds at points A and B in the figure where:

$$OSC/(K*M) = INPUT/N = COMPARISON\ FREQUENCY \quad (EQ\ 1)$$

This technique is based on selecting a reference oscillator frequency that is slightly higher than OSC. This frequency (OSC+) should be chosen so that:

$$1/COMPARISON\ FREQ. - (K*M)/(OSC+) = .5 * (1/OSC) \quad (EQ\ 2)$$

The right side of EQN. 2 equals one half the period of the reference oscillator.

The reference oscillator frequency delta will cause point B (the detector flip-flop D input) to begin to precede point A (the detector flip-flop clock input) by half a period. When the edge of the D input is sufficient the detector will clock true and begin a pulse train through the two de-glitching flip-flops. The output of the second of these clears all three flip-flops and steals a pulse by disabling the divide by K output. Stealing the pulse puts point B behind A until the reference oscillator delta can move it ahead by one period repeating the cycle. Points A and B are always within one half a cycle of each other.

The circuit allows the frequency of the output signal to be selected simply be adjusting the values of the dividers K and M. The lock range of the loop is given by the following:

$$Lock\ Range = +- (OSC+/osc)/INPUT \quad (EQ\ 3)$$

Jitter Bounded PLL

Another technique [Walters] for generating a wide variety of synchronized clock frequencies with low jitter employs an accumulator Digital Controlled Oscillator (DCO) and phase and frequency comparators. The system, shown in Figure 3.6.27 , can lock to any division of a reference frequency (F_{ref}), as selected by the data loaded into frequency divider counters.

The Successive Approximation Register (SAR) and its controller serves as a low-pass filter supplying the DCO with frequency and phase correction data. Among the three inputs to the SAR controller is the F_{ref} divided by a factor Q to form F_q.

The other two inputs come from the phase and frequency (zero) comparators. The frequency comparator output is the DCO frequency divided by P to form F_p. When the system is in lock the following equation is true:

$$F_{dco} = (P/Q) * F_{ref}. \quad (EQ\ 4)$$

The heart of the system is the accumulator DCO which determines the ability to lock to a frequency and the amount of jitter allowed. The DCO consists of a four-bit accumulator whose input is fed by the SAR. The DCO input value is determined from

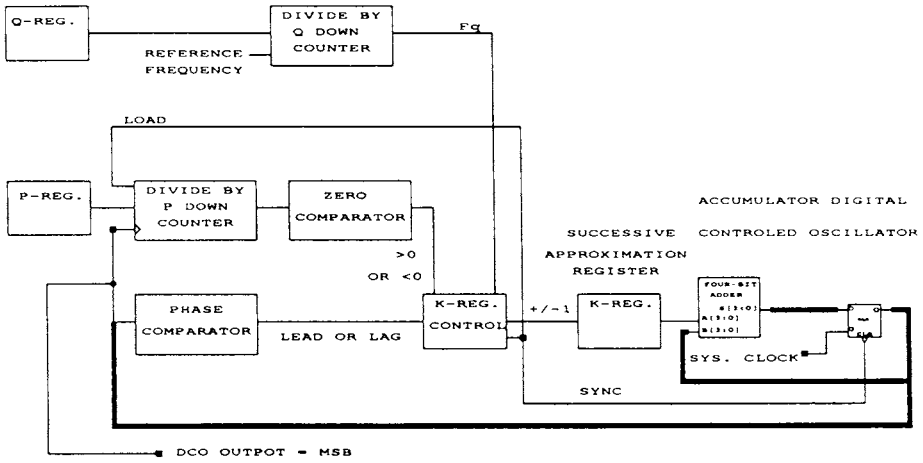


Figure 3.6.27. Jitter Bounded Digital PLL

the phase and frequency comparison feedback loops. The most significant bit of the accumulator output is the DCO output signal. It is generated by successively adding the SAR value to itself at the high- frequency system clock rate. The frequency comparator uses the value of P to divide the DCO frequency. If the frequency is out of lock during a period of $F_{ref.}$, the comparator asserts greater-than-zero or less-than-zero to the SAR controller to modify the value of the register. If the P counter output is zero, the DCO has the correct frequency.

The DCO latch acts as a phase register indicating the phase of the DCO with respect to F_{ref}. The DCO phase is calculated by the N most significant accumulator output bits. When the DCO is out of phase the jitter, or phase difference, is detected by the phase comparator and accumulates with time until it equals one period. The feedback

loops then cause the SAR register controller to load a correcting value into the register or to clear the accumulator with a synchronizing pulse.

The Jitter-Bounded DPLL may be implemented entirely on an ACT FPGA. The resource requirements vary with the relationships of the system input and output frequencies, but for any F ref, system clock, and desired output frequency the design is easily accommodated on an ACT FPGA.

Customer Design Examples

Example 1: Functional Prototyping

A multi-processor file server for a PC network is under development. The system employs multiple 80486 microprocessors, banks of DRAM, and multiple 64- and 32-bit busses. Several 1280s FPGAs are used to prototype the system which will ultimately be implemented with gate arrays. Since the logic was composed of clearly defined sub-blocks, principally state machines and simple data paths, partitioning of the logic among the FPGAs was not difficult. For this complex system, the design process is incremental and somewhat experimental. The logic in each FPGA may be modified or extended several times, so a fast design flow is necessary.

The logic for each FPGA is synthesized with the Synopsys hardware description language (HDL) compiler using an Actel macro library, and then checked with an HDL simulator. Each design is automatically placed and routed, and re-simulated with post-layout delays. Although the system will ultimately operate at a high speed, use of synchronous logic design allows the prototype to operate at a speed within the capability of the FPGAs. Software is developed and tested on the prototype concurrently with the completion of the logic design and conversion to conventional gate arrays.

Example 2: At-Speed Prototyping and Pilot Production

A digital video signal processing system in high volume production for the consumer market. Four 1020 FPGAs were used to integrate registers, multipliers, adders and digital phase- locked-loops. The pipelined data paths are 12 to 16 bits wide and operate at a 16MHz clock rate. For production the four FPGAs were replaced by a single 8000 gate masked gate array. It is estimated that the use of FPGAs for prototyping and pilot production saved two to three months in product development time.

As many as half of masked gate array designs never reach high-volume production. The reasons include short product lifetimes, required new features or other changes, or simple lack of demand for the product. In such cases use of FPGAs for production saves the NRE costs associated with developing a mask programmed gate array.

For very large volumes, the design may be transferred to a conventional gate array. Due to the small granularity of the module, an FPGA design can be efficiently

converted to a gate array. Test vectors can often be generated automatically. Several vendors support this conversion path.

3.7 Acknowledgments

This chapter includes information compiled by many people. In particular Jonathan Greene and Esmat Hamdy contributed portions of this chapter. Sam Beal, Warren Miller and Jeff Schlageter both contributed material and made suggestions about the presentation.

3.8 References

[Ahrens] M. Ahrens, et. al., *An FPGA Architecture Optimized for High Densities and Reduced Routing Delay,* Proceedings Custom Integrated Circuits Conference, July1990.

[Allen] D. Allen, R. Goldenberg, *Design Aids and Test Results for Laser-Programmable Logic Arrays,* Proceedings. International Conference. on Computer Design, 1990, pp. 386-390.

[Birkner] J. Birkner et al., *A Very High-Speed Fleld Programmable Gate Array Using Metal-to-Metal Antifuse Programming Elements,* Proceedings Custom Integrated Circuits Conference, May 1991.

[Brayton] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, A. Wang. *MIS: A Multiple-Level Logic Optimization System.* IEEE Transactions on CAD, Nov. 1987.

[Carter] W. Carter et al, *A User Programmable Reconfigurable Gate Array,* in Proceedings of Custom Integrated Circuits Conference, pp. 233-235, 1986.

[Chen] J. Chen et al, *A Modular 0.8 um Technology for High Performance Dieletric Antifuse Field Programmable Gate Arrays,* International Symposium on VLSI Technology, Systems, and Applications, May 1993, pp. 160-164.

[Chiang1] S. Chiang, R. Wang, J. Chen, K. Hayes, J. McCollum, E. Hamdy, C. Hu. *Oxide-Nitride-Oxide Antifuse Reliability,* International Reliability Physics Symposium., March 1990, pp.186-192.

[Chiang2] S. Chiang, K. Hayes. *Act 1010/1020 Reliability Report.* Actel Corporation, Sunnyvale, CA, April 1990.

[ElAyat] K. El Ayat, et. al. *A CMOS Electrically Configurable Gate Array.* IEEE J. Solid-State Circuits, Vol. 24, No. 3, June, 1989, pp. 752-762.

[ElGamal1] A. El Gamal, J. Greene, J. Reyneri, E. Rogoyski, K. El-Ayat, and A. Mohsen. *An Architecture for Electrically Configurable Gate Arrays.* IEEE J. Solid-State Circuits, Vol. 24, No. 2, April, 1989, pp. 394-398.

[ElGamal2] A. El Gamal, J. Greene, V. Roychowdhury. *Segmented Channel Routing*

is Nearly as Efficient as Channel Routing (and Just as Hard). Proceedings. VLSI Conference, Santa Cruz, CA, March 1991.

[ElGamal3] A. El Gamal. *Two Dimensional Stochastic Model for Interconnections in Master Slice Integrated Circuits.* IEEE Trans. on Circuits and Systems, CAS-28, 127-138, Feb. 1981.

[Gerzberg] L. Gerzberg. U.S. Patent 4,590,589, 1986.

[Graham] H. Graham, D. Seltz. *Electronically Programmable Gate Array Having Programmable Interconnect Lines,* U.S. Patent 4,786,904, Nov. 22, 1988.

[Greene] J. Greene, V. Roychowdhury, S. Kaptanoglu, A. El Gamal, *Segmented Channel Routing,* Proceedings. Design Automation Conference., Orlando, Florida, Association. for Computing Machinery, June 1990.

[Hamdy] E. Hamdy, J. McCollum, S. Chen, S. Chiang, S. Eltoukhy, J. Chang, T. Speers,. Mohsen. *Dielectric Based Antifuses for Logic and Memory ICs,* IEDM Tech. Digest, pp. 786-789, 1988.

[Hashimoto] A. Hashimoto, J. Stevens. *Wire Routing by Optimizing Channel Assignment within Large Apertures.* Proceedings. 8th IEEE Design Automation Workshop, 1971.

[Holmberg], et al. U.S. Patents 4,499,557, 1985 and 4,599,705, 1986.

[Hsieh] H. Hsieh, et. al., *Third-Generation Architecture Boosts Speed and Density of Field-Programmable Gate Arrays,* Proceedings. 1990 Custom Integrated Circuits Conference., May 1990, pp. 31.2.1-32.2.7.

[Karplus] K. Karplus, *AMAP: a Technology Mapper for Selector-based Field-Programmable Gate Arrays,* Proceedings of the 28th Design Automation Conference, June 1991, pp. 244-247.

[Lim] Lim et al. U.S. Patent 4,569,121, 1986. Stacy et al. U.S. Patent 4,569,120. 1986.

[Lorenzetti] M. Lorenzetti, D. Baeder. *Routing.* Chapter 5 in Physical Design Automation of VLSI Systems, B. Preas and M. Lorenzetti, eds., Benjamin Cummings, 1988.

[LSI] *HCMOS Gate Array Databook and Design Manual,* LSI Logic Corporation, October 1986.

[Mailhot] F. Mailhot, G. De Micheli. *Technology Mapping Using Boolean Matching and Don't Care Sets.* preprint, 1990.

[Marr] C. Marr, *Logic Array Beats Development Time Blues,* Electronic System Design Magazine., Nov. 1989, pp. 38-42.

[Murgai] R. Murgai, Y. Nishizaki, N. Shenoy, R. Brayton, A. Sangiovanni-

Vincentelli. *Logic Synthesis for Programmable Gate Arrays.* Proceedings. 27th ACM/IEEE Design Automation Conference., 1990.

[Preas] B. Preas, P. Karger. *Placement, Assignment and Floorplanning.* Chapter 4 in Physical Design Automation of VLSI Systems, B. Preas and M. Lorenzetti, eds., Benjamin Cummings, 198.

[Roesner] B. Roesner. U.S. Patents 4,424,579 and 4,442,507, 1984.

[Rose] J. Rose, R. Francis, D. Lewis, P. Chow. *Architecture of Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency.* IEEE Journal of Solid State Circuits, Vol. 25, No. 5, pp. 1217-1225, October 1990.

[Rudell] R. Rudell, R. Segal. *Logic Synthesis Can Help in Exploring Design Choices,* 1989 Semi-custom Design Guide, CMP Publications, Manhasset, NY.

[Schlageter] J. Schlageter et all, *An Advanced Sub-Micron Architecture for IO Intensive Applications,* Proceeding of the 1993 Compcon conference, 1993, pp. 362-366.

[Singh] S. Singh, J. Rose, D. Lewis, K. Chung, P. Chow, *Optimization of Field-Programmable Gate Array Logic Block Architecture for Speed,* Proceedings of the 1991 CICC Conference, 1991, pp. 6.1.1-6.1.6.

[Smith] J. F. Smith, et. al., *Laser-Induced Personalization and Alterations of LSI and VLSI Circuits,* Proceedings of. 1st International Laser Processing Conference., Anaheim, Calif., Laser Institute of America, Nov. 16, 1981.

[Stopper] H. Stopper, et al. U.S. Patent 4,847,732, 1989.

[Walters] Walters, S., Troudet, T.; *Digital Phase-Locked Loop with Jitter Bounded.,* IEEE Transactions on Circuits and Systems, VOL. 36, NO. 7, July 1989.

[Whitney] T. Whitney and J. Schlageter, *A New High Performance Field Programmable Gate Array Family,* Proceedings of 1993 International Conference on Computer Design, October 1993.

[Whitten] R. Whitten, R. Bechtel, M. Thomas, H.T. Chua, A. Chan, J. Birkner, European Patent Application No. 90309731.9, May 9, 1990.

[Wong] S. Wong, H. So, J. Ou, J Costello, *A 5000-Gate CMOS EPLD with Multiple Logic and Interconnect Arrays,* Proceedings. 1989 Custom Integrated Circuits Conference., May 1989, pp. 5.8.1-5.8.4.

Chapter 4
Erasable Programmable Logic Devices

The Technical Staff of Altera Corporation
Edited by Robert Hartmann

4.1. Introduction

Various acronyms are used to describe programmable logic, such as FPGA, GAL, PAL, EPLD, FPLA. Several of these acronyms have come to imply a particular architecture. EPLD, for example, is usually identified with array-based (AND-OR) programmable logic. This structure is particularly well-suited for implementing wide fan-in logic functions. These array-based structures were popularized in several families of bipolar, fuse-programmable logic devices but most particularly the PAL [Birkner 1978] which was introduced by MMI (Monolithic Memories, Inc. which was later acquired by Advanced Micro Devices.)

While these PAL devices were low in complexity by today's standards, they allowed the integration of several SSI and MSI components into one device. This integration advantage combined with high speed, flexibility to do both combinational and sequential functions, reasonable cost and the availability of software tools made these devices extremely popular. But, there were also drawbacks. The two most significant problems were high power dissipation and fuse programmability which meant that incorrectly programmed devices could not be salvaged. These disadvantages ultimately limited the logic complexity to a few hundred gates.

The EPLD (Erasable Programmable Logic Devices) developed by Altera overcame the limitations of bipolar fuse-programmed PALs by using CMOS technology in conjunction with floating-gate programmable transistors. This technology opened the way for reprogrammable circuits with much higher logic capacity and much lower power.

Altera's first architecture, known as Classic, was introduced in 1984 and offered chips ranging from 8 to 48 macrocells (see P. 7), which roughly translates to 300 to 2,000 gates [Hartmann 1984]. The Company's second architecture, introduced in 1988 and designated the MAX 5000 family, provides from 32 to 192 macrocells or 600 to 7,500 gates. In 1991 a third-generation architecture, called the MAX 7000 family, was introduced. The first member, the EPM7256, has 256 macrocells (approximately

10,000 gates). Devices with as many as 1,000 macrocells are planned. Each new generation has provided faster performance and greater gate density than preceding generations.

Proprietary software tools that make the use of EPLDs quick and simple were also developed. These software tools improve the productivity of the engineer, and can be used repeatedly by multiple users for different designs on an ongoing basis.

An EPLD logic design for a particular end use is achieved in three steps: design entry, design compilation, and design verification.

Design entry is accomplished using either the integrated software editors that are part of Altera's software, or -- if the customer prefers -- with tools provided by EDA vendors. Designs can be expressed as schematic diagrams, a text-based hardware description language, logic equations, state machines, truth tables, waveforms, or a combination of any of the above.

Design compilation of the design is performed by software that first "synthesizes" the logic specified during design entry so that the minimum number of logic cells (sometimes referred to as macrocells) are used. The software then arranges and interconnects these logic cells in the most compact manner possible (this is commonly called "fitting"). With early PLDs, logic synthesis was nothing more than factoring Boolean equations into the minimum number of product terms. If that number of generated product terms was less than or equal to the number available, then the "synthesis" was complete. If not, the user had to restructure the logic and try again.

Logic may be described in a variety of ways (gates, logic equations, hardware description language, etc.), all of which may be intermingled in the software environment. The job of logic synthesis is to take the logic and reduce it in such a way that it uses the minimum of a given device's resources. The synthesizer must have knowledge of the device's architecture and a set of minimization techniques in order to accomplish the task. In the Altera software, as many as eighteen minimization methods will be tried, with the result of the trials compared before the best implementation is selected. The proprietary algorithms that perform this function require minimal intervention from the user and offer highly-automated, "push-button" results.

Finally, in design verification, the designer confirms the logic functions performed by the EPLD and checks the timing of critical logic paths. Designs are verified using the static Timing Analyzer, Delay Predictor, and logic Simulator (which has detailed timing models for all of the internal EPLD elements), all of which are part of the design software tool kit.

4.2. Programming Technology

Prior to the introduction of the EPLD, the only technology used for Programmable Logic Devices (PLDs) was bipolar and fuse-based. The active elements on these devices were bipolar transistors with arrays of fuses providing programmable interconnect structures. These fuse elements consisted of a variety of exotic metal alloys and/or polysilicon structures, but all relied on the physical destruction of fuses to open connections by passing large currents through their small geometries.

The melting process in bipolar PLD fuses was difficult to control and often resulted in unacceptable programming yields. Since the process was irreversible, guaranteed results were impossible. The power-hungry bipolar technology also severely limited integration levels. With the advent of the EPLD, CMOS technology replaced bipolar technology, and fuses were replaced by reprogrammable EPROM or EEPROM transistors (EPLD bits). These EPLD bits were much smaller than fuses, electrically programmable, and erasable. Since the programming step was reversable, EPLDs were fully factory-tested, guaranteeing 100% programming yield at the customer site. The lower power required of CMOS technology allowed higher integration levels.

The EPROM cells operate via floating-gate charge injection [Wolf 1990]. The programming process consists of placing sufficient voltage (typically >12V) on the drain and gate of the transistor to create a strong electric field and energize electrons to jump from the drain region to the floating gate. Electrons attracted to the floating gate become trapped when the high voltage bias is removed. If the drain terminal is held at a low voltage during programming, electrons are not available to be attracted to the floating gate and the floating gate remains uncharged. Trapped charge changes the threshold of the EPROM cell from a relatively low value with no charge present ("erased") to a higher value when programmed. Figure 4.2.1 shows a basic cross-section of the EPROM cell technology.

The topographical view of a single EPROM cell is shown in Figure 4.2.2. The "Poly Select Line" in Figure 4.2.2 corresponds to the terminal "GATE" in Figure 4.2.1; the "Contact Area" and "Ground Diffusion" correspond to the "Drain" and "Source" regions of the EPROM transistor respectively.

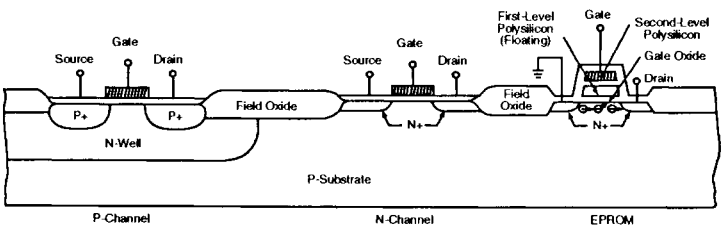


Figure 4.2.1 Cross Section of CMOS EPROM Die

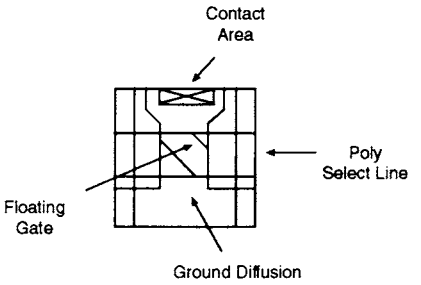


Figure 4.2.2 Top View of Single EPROM Transistor

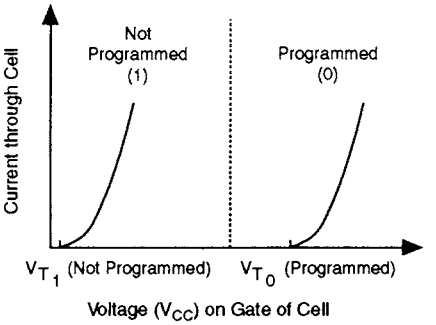


Figure 4.2.3 V-I Characteristic of EPROM Transistor

The EPROM transistor has two states, erased (unprogrammed) and programmed. The V-I characteristics for these two states is shown in Figure 4.2.3.

If it is assumed that the floating gate in the structure is initially unprogrammed, then the transistor works much like a normal N channel transistor. Whenever a positive voltage greater than the threshold voltage (VT1) is applied to the control gate, a channel is induced under the gate region which allows current to flow between the drain region and the source region. In a typical N channel floating gate device, VT is approximately 1.0 volts. The signals which are applied to the control gate are typically between 0 volts and 5 volts.

The effective threshold of the transistor can be changed as explained previously. After programming, charges remain trapped on the floating gate and cause the effective threshold of the EPROM transistor to be increased to a value greater than the voltage which would be applied to the control gate during Normal Operation. The application of the high voltages and the subsequent charge trapping on the floating gate is called programming.

After the transistor has been programmed, if a 5 volt signal is applied to the control gate, no channel will be created between the source and drain and no current will flow. This two layer transistor can be thought of as a programmable switch. In the unprogrammed state, the switch opens and closes in response to the application of 0 volts or 5 volts to the control gate. In the programmed state, the switch is always open regardless of whether 0 or 5 volts is applied to the control gate.

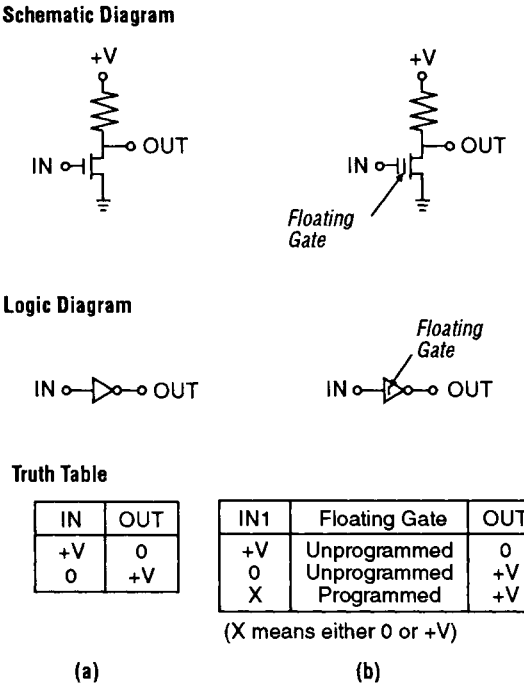
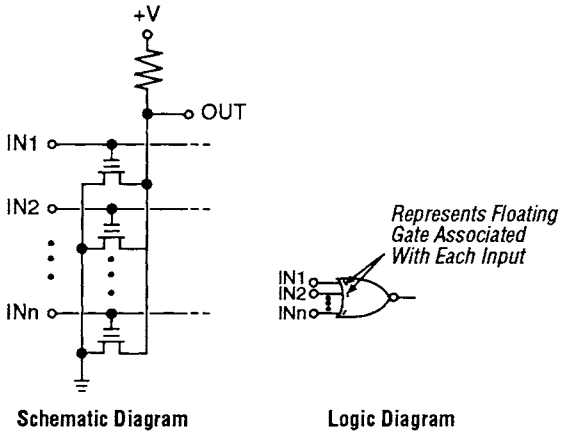


Figure 4.2.4 Use of an EPROM Transistor in a Programmable Inverter

Logic Structures Using EPROM Transistors

In the diagrams shown in Figure 4.2.4, an inverter is shown in part (a) using a standard N-channel transistor as the switching device and, in part (b) a floating gate transistor is shown as the switching element. Below each schematic diagram, a logic gate representation and the corresponding truth table is shown. The truth table for the EPROM inverter in the unprogrammed state, gives the same result as the normal N-channel inverter. However, when the floating gate is programmed, the output is always pulled high, independent of the input. Thus, a logic function (an inverter in this case) can be created wherein the input variable can be "programmed out" by programming the floating gate to which the input is connected.



Truth Table				
IN1	IN2	INn	OUT	Note: P in the Truth Table means that the EPROM transistor at that site is programmed. X means either 1 or 0.
0	0	0	1	
X	X	1	0	
X	1	X	0	
1	X	X	0	
X(P)	0	0	1	IN1 Gate is programmed
X(P)	1	X	0	
X(P)	X	1	0	
0	X(P)	0	1	IN2 Gate is programmed
1	X(P)	X	0	
X	X(P)	1	0	
0	0	X(P)	1	INn Gate is programmed
1	X	X(P)	0	
X	1	X(P)	0	

Figure 4.2.5 Programmable NOR Gate Using EPROM or EEPROM Transistors

In part (a) of Figure 4.2.5 a NOR gate circuit using EPROM elements is illustrated. In this circuit, the output is only a function of the inputs corresponding to floating gates left in the erased state. In part (b), the corresponding logic diagram is shown, and in part (c) the truth table for the circuit is shown.

This circuit is a one dimensional array of EPROM elements which forms a NOR gate, the elements of which are selectively programmable. By extending this concept to a two dimensional array and then collecting the programmable NOR outputs in another set of NOR gates, electrically programmable logic arrays such as illustrated in Figure 4.2.6 can be formed.

Any combinational logic function, however complex, can be expressed as a Boolean

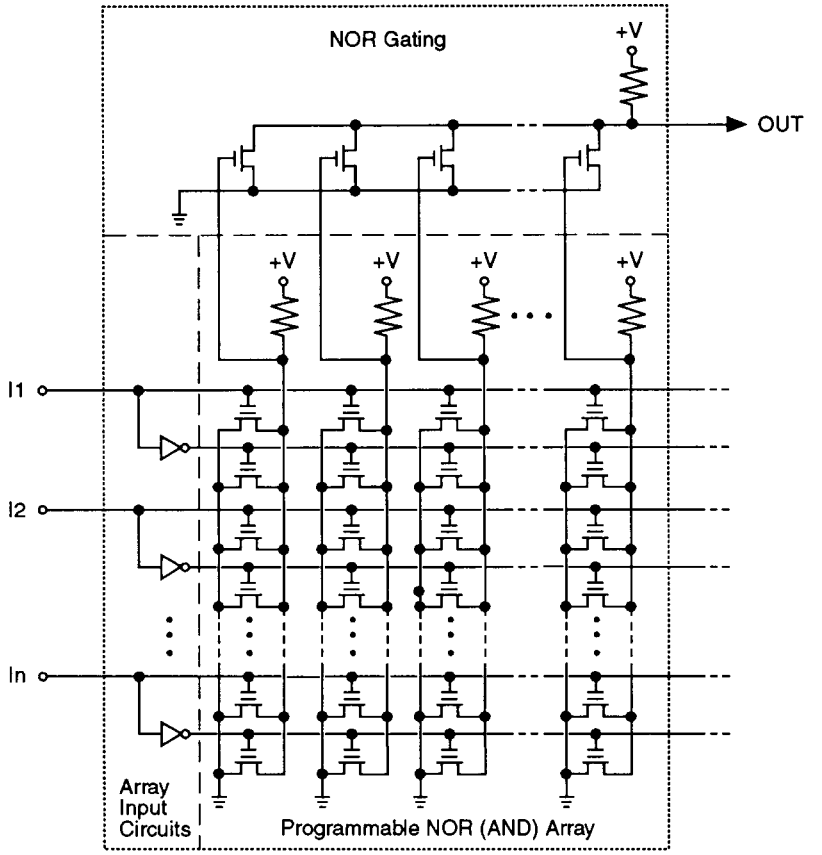


Figure 4.2.6 Programmable NOR Array

equation. Furthermore, any Boolean equation can be expressed in sum-of-products form. The logic expression is of the form f=P1+P2+P3+... where f is some Boolean variable and P1, P2, etc. are product terms. Product terms can be expressed as P = A1*A2*A3.... That is, product terms are the logical AND of literals. The hardware realization consists of an array (whose inputs are the true and complement of the literals) which generates the product terms and another array (or sometimes a simple OR gate) which generates the sum-of-product expression. Sequential functions are accommodated by adding a register (such as a D flip-flop). The input to the register is fed by the sum array output and the output of the register is fed back as an input to the product term array.

The NOR array shown in Figure 4.2.5 can be thought of as a product term generator and the sum-of-products can be generated by a structure like that shown in Figure

4.2.6. The connection from a particular input variable to a product term is a function of the state of the EPROM transistors.

Initially all of the EPROM transistors in the array are erased and are therefore connected to all of the product terms. By selectively programming these EPROM transistors, certain of the inputs are disconnected, leaving only those variables which are needed by a particular logic expression.

Each EPLD contains one or more AND arrays that provide product terms. A product term is simply an n-input AND gate, where n is the number of variables. EPLD schematics use a shorthand AND-array notation to represent large AND gates with common inputs. Figure 4.2.7 shows three different representations of the same logic function. Circuit A is presented in classic logic notation; Circuit B has been modified to a sum-of-products notation; and Circuit C is written in AND-array notation. A dot represents a connection between an input (vertical wire) and one of the 8-input AND gates. No dot implies no connection. Unused AND gate inputs float to a logic 1.

Even the rather simple 2X8 AND-array of Circuit C can produce all one and two variable functions, most three variable and some four variable functions. The table below shows the number of possible functions of one, two, three and four variables

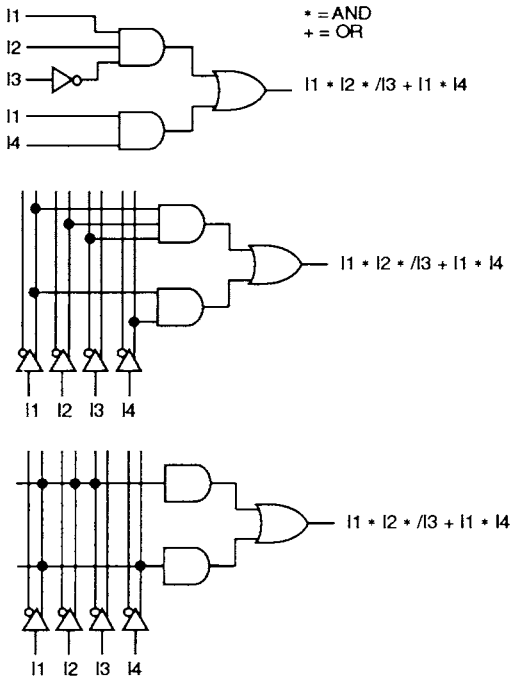


Figure 4.2.7 AND-Array Notation

and the ability of a two product-term AND-OR structure to synthesize these functions.

Number of Variables	Number of Functions Possible	Number of (1) Unique Functions	Coverage (2 product terms)
1	4	2	All
2	16	4	All
3	256	14	9 of 14
4	65,536	222	18 of 222

(1) Allows for inversions and reorderings of inputs and inversion of the output.

4.3. Device Architecture

Up to this point we have been discussing EPLDs at the level of transistors, EPROMs and product terms in order to provide the reader with some of the basic fundamentals. However, the use of software tools can eliminate the necessity of understanding any of the circuit-level complexities of EPLD architectures. The user may then work with familiar design entry tools (e.g., TTL macrofunctions or a high-level design language), and the software can automatically translate the design into the format required to fit the EPLD architecture.

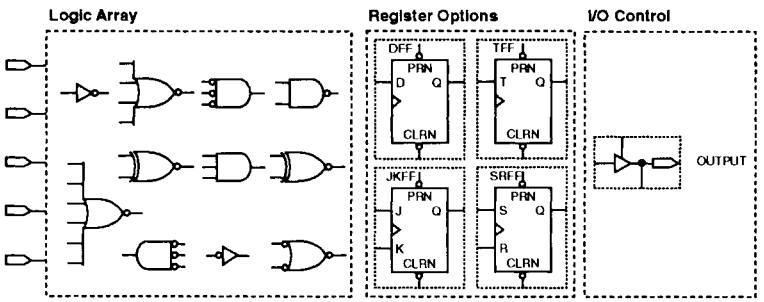


Figure 4.3.1 Basic Logic Resources of an EPLD

Basic Concepts

The heart of the EPLD is the logic array as discussed previously. In addition, general-purpose EPLDs contain a number of other resources, such as dedicated input pins, user-configurable I/O pins, and programmable flip-flop and clock options that maximize flexibility for integrating random logic functions. Figure 4.3.1 depicts a logic designer's view of the logic resources of a typical EPLD.

Input variables to the logic array come from the input and I/O pins and from the output of macrocells. The logic array outputs drive output pins and inputs to the flip-flops.

Macrocell Architecture

The fundamental logic building block of an EPLD is the macrocell. Each macrocell consists of three parts:

- The logic array which implements all combinational logic functions.
- The programmable register which can be configured to provide D, T, JK, or SR options (the register can also be bypassed).
- Programmable I/O. Each I/O pin can be configured as a dedicated output, a dedicated input, or as a bidirectional pin.

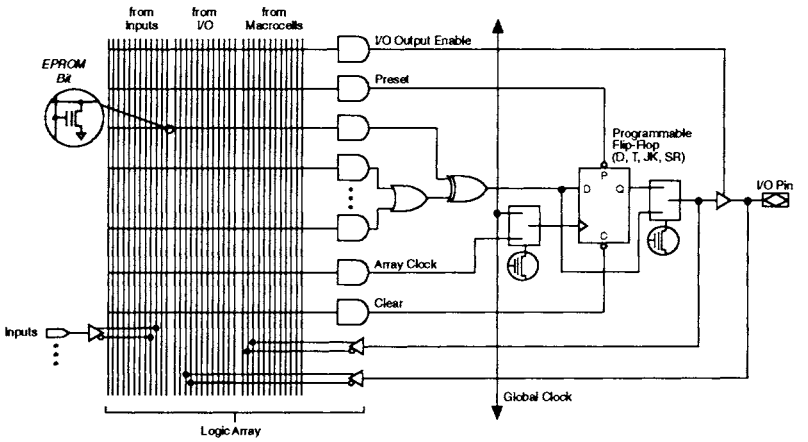


Figure 4.3.2 Typical EPLD Macrocell Logic Diagram

Logic Array

Figure 4.3.2 is a diagram of a typical EPLD macrocell.

The logic array is a programmable-AND/fixed-OR programmable logic array. The inputs to the AND array come from dedicated input pins, I/O pins, and from macrocell feedback paths. Typically, both true and complement versions of any of these signals are available as inputs to the programmable AND gates.

The macrocell shown in Figure 4.3.2 contains product terms that can be used for both combinational and sequential functions. Connections between the array inputs and the product terms are created during the programming process. One may think of a

product term as an AND gate with many possible true/complement input signal pairs. Any product term may be connected to the true or complement (or both or neither) of any array input signal. If both the true and complement of any input signal are left intact, a logical false results on the output of the product term. If both the true and complement connections of any input signal are open, a logical "don't care" results for that input. If all inputs for the product term are programmed open, a logical true results on the output of the product term.

Several product terms feed a fixed OR whose output connects to an exclusive-OR (XOR) gate. The second input to the XOR function is controlled by a programmable resource (usually a product term) that allows the logic array output to be inverted. Software products which support this architecture can take advantage of this XOR function to implement active-high or active-low logic, complex mutually exclusive and arithmetic functions, or to reduce the number of product terms needed to implement a function (by applying De Morgan's inversion). Figure 4.3.3 illustrates an example of an OR function that requires six product terms in its original form. By using the "programmable" XOR gate and De Morgan's inversion, the OR function can be transformed into a NAND function:

A+B+C+D+E+F = / (/A*/B*/C*/D*/E*/F)

This inversion from OR to AND allows the equation to be implemented in a single product term.

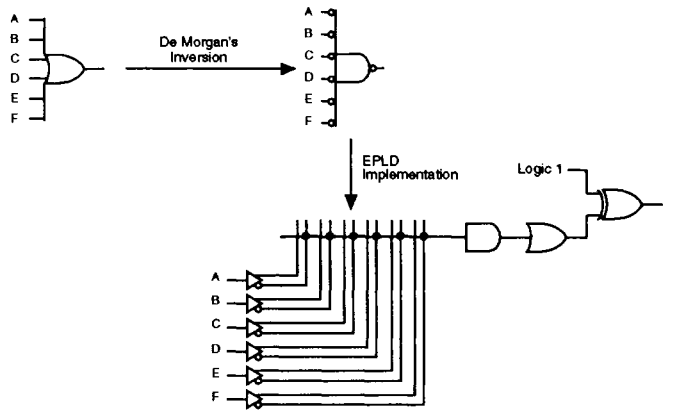


Figure 4.3.3 Product Term Reduction Using De Morgan's Inversion

Programmable Flip-Flops

Programmable flip-flops are used to create a variety of logic functions that use a minimum of EPLD resources. Each flip-flop can be programmed to provide a conventional D, T, JK, or SR function. In some devices, the macrocell flip-flops can

also be configured as flow-through latches. Macrocell flip-flops may also have an asynchronous Clear and Preset capability which allows complete emulation of many commonly used TTL macrofunctions.

Programmable Clock

In many PLD architectures, the clock source for the macrocell flip-flop is programmable. Each flip-flop may be clocked from a dedicated device input pin (also known as a synchronous or global clock), or from any input or I/O pin, or any internal logic function (via a product term). The product term clock source is often called the array clock. A clock selection circuit associated with each flip-flop is programmed to make the desired choice. Flip-flops can thus be clocked independently or in user-defined groups. Macrocell flip-flops are typically positive-edge-triggered with data transitions that occur on the rising edge of the clock. Array clocks allow positive or negative edge triggered clocks, gated-clocks and clock-enable logic to be implemented. However, global clock signals have faster clock-to-output delay times than internally-generated product term clock signals.

I/O Control Block

Figure 4.3.4 illustrates the resources associated with a typical I/O pin. The I/O block contains a tri-state buffer whose data input comes from a macrocell. The tri-state function is controlled by a macrocell product term. I/O pins may be configured as dedicated outputs, as bidirectional outputs, or as additional dedicated inputs. In most PAL architectures (PAL 16R8 or 22V10, for example) there is a physical coupling of I/O pins to macrocells. In such architectures, when an I/O pin is used as an input, the macrocell associated with that pin cannot be used because the macrocell feedback path is used by the I/O input signal. This is a waste of a valuable resource. More recent architecture (e.g., MAX 5000 and MAX 7000) have "dual feedback", whereby the macrocell feedback is decoupled from the I/O pin feedback. Dual feedback makes it possible to implement a "buried" function in the macrocell while the I/O pin is used simultaneously as a dedicated input. Applications that require many buried flip-flops such as counters, shift registers, and state machines, or bus-oriented functions are more easily accommodated by this type of programmable I/O block.

Design Security

All EPLDs contain a programmable Security Bit that controls access to the data programmed into the device. If this feature is used, a proprietary design implemented in the device cannot be copied or retrieved. This feature provides a high level of design security, since programmed data within EPROM or EEPROM cells is invisible. The Security Bit that controls this function, as well as all other program data, is reset by erasing the EPLD.

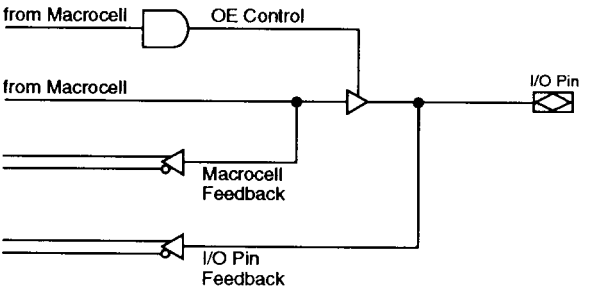


Figure 4.3.4 EPLD I/O Pin with Dual Feedback

Functional Testing

Because they are erasable and reprogrammable, EPLDs may be fully tested, and conformance to specification may be guaranteed. Complete testing of each programmable EPLD bit and all internal logic elements ensures 100% programming yield. Test programs may be used and then erased during early stages of the production flow. This facility to use application-independent, general-purpose tests, called generic testing, is unique to reprogrammable user-configurable logic devices. EPLDs also contain on-board logic test circuitry to allow verification of function and AC specifications of devices in windowless packages.

Operating Requirements for EPLDs

Most devices operate at a nominal power supply voltage of 5 volts (some 3.3 volt parts are also available). Input levels and output drive characteristics are consistent with TTL family characteristics.

Certain precautions are required for trouble-free operation. Unused inputs must be tied to VCC or GND. Unused I/O pins should also be tied to VCC or GND. Each set of VCC and GND pins must be decoupled directly at the device.

Whenever many output pins are switching simultaneously, system noise can be generated. This noise is usually seen on the Vcc supply or on adjacent quiescent outputs. In severe cases, this noise can cause unwanted triggering of flip-flops, either on the device itself or on other devices in the system. Evolving circuit design techniques have done much to eliminate the problem. However, faster slew rate outputs and higher drive capability of new devices make the circuit design problem ever more difficult. As a general precaution whenever a design allows eight or more outputs to switch simultaneously, special precautions in PC board design are recommended. Among these are: use of embedded Vcc and GND planes, scrupulous use of decoupling capacitors, restriction of signal trace length to eight inches or less, and use of small series resistors (10 ohms to 30 ohms) on long traces or those with highly capacitive loads.

Architectural Evolution in Array-Based PLDs

In the preceding sections the general features of array-based architectures have been discussed. The evolution began with the basic programmable AND array whose outputs feed either a fixed OR (the PAL structure) or a programmable OR (the FPLA structure) to produce sum-of-products logic implementation. Flip-flops were added to this structure to allow implementation of sequential functions. Many ancillary functions were then added to provide for better utilization of the parts. More flexible I/O structures, programmable flip-flop clock control, programmable flip-flop types (e.g., D, J-K, T) are examples of these enhancements. However, these early parts beginning with the PAL (MMI) and continuing through the 22V10 (AMD), the EPxxx series (Altera) and the GAL (Lattice) all were single array, globally connected parts. "Globally connected" means that the output signal(s) of every macrocell is fed back as an input to the logic array.

A fundamental problem with simple expansion of macrocell-based PLDs is that as the number of macrocells grows by N, the programmable elements (e.g., EPROM or EEPROM bits) in the array which feeds those macrocells grows by N-squared [Wong 1989]. There are two problems with increasing the bit count: first, the yield suffers because of the larger die size, and second, the delay through the array increases. Much of the architectural work that has gone on in array-based PLDs has been directed at breaking the N-squared array growth relationship to increasing complexity. It has been a primary focus of research at Altera, AMD, Lattice and others. In the sections that follow, we will explore the architectural evolution using three generations of products developed at Altera. As we look at each product, we will try to point out the problems that existed and the architectural solution that was implemented. While this treatment is not exhaustive, the discussion of these product architectures will provide a good basis for understanding the product architectures of other vendors.

4.3.1 - The "Classic" Family of PLDs

The original family of Altera EPLDs are grouped into what is known as the "Classic" family. The largest member of this family is the EP1810. The architecture used in the EP1810 is an early attempt at breaking single, large, globally-connected arrays into a number of smaller "local" arrays with a connection structure between these local arrays. An understanding of the EP1810 architecture provides a good foundation for understanding the more complex MAX 5000 and MAX 7000 products to be described in a later section.

Functional Description of the EP1810

Figure 4.3.1.1 shows the complete block diagram of an EP1810 EPLD. The EP1810 device has four identical quadrants, each containing 12 macrocells. Internal bus structures in these EPLDs feed input signals into the macrocells. Macrocell outputs drive the external pins and internal buses.

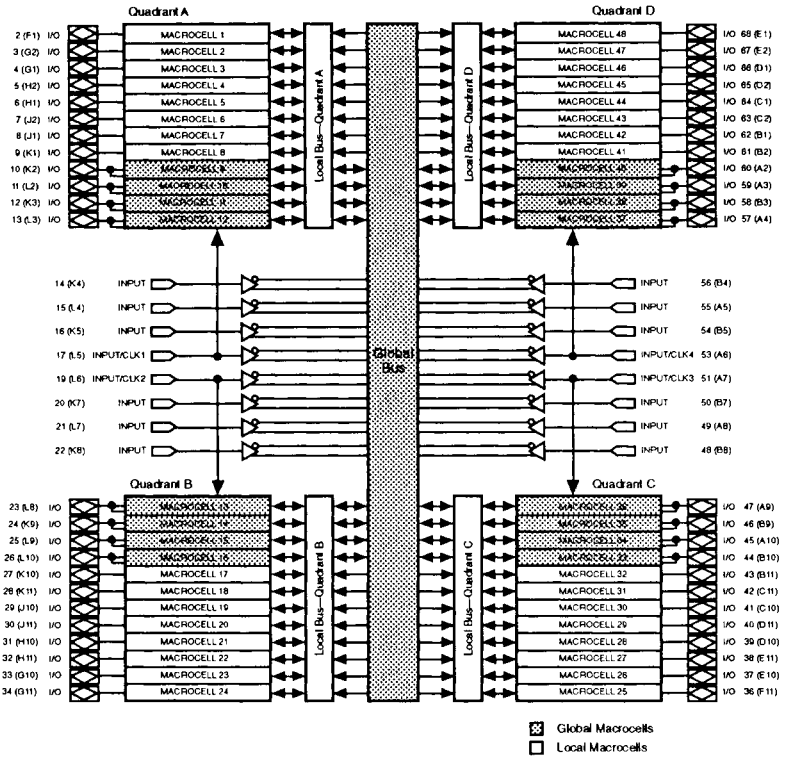


Figure 4.3.1.1 EP1810 Block Diagram

EP1810 EPLDs use CMOS EPROM cells to configure internal logic functions. The architecture is 100% user-configurable, accommodating a variety of independent logic functions. Figures 4.3.1.2 and 4.3.1.3 depict the detailed macrocell architecture used in this device.

EP1810 EPLDs have 48 macrocells, 16 dedicated data inputs, 4 global Clock inputs, and 48 I/O pins that can be individually configured for input, output, or bidirectional operation on a macrocell-by-macrocell basis. Each macrocell contains 10 product terms for the following functions: 8 product terms are dedicated to logic implementation; 1 product term is used for Clear control of the internal register; and 1 product term implements either Output Enable or an array Clock.

Of the 48 macrocells, 32 are local (see Figure 4.3.1.2) and 16 are global macrocells (see Figure 4.3.1.3). Local macrocells offer a multiplexed feedback path (with pin or macrocell feedback) and drive the local bus in their quadrant. Global macrocells feature two dedicated feedback paths: one feeds the local bus; the other feeds the global bus. This arrangement, called "dual feedback," allows global macrocells to

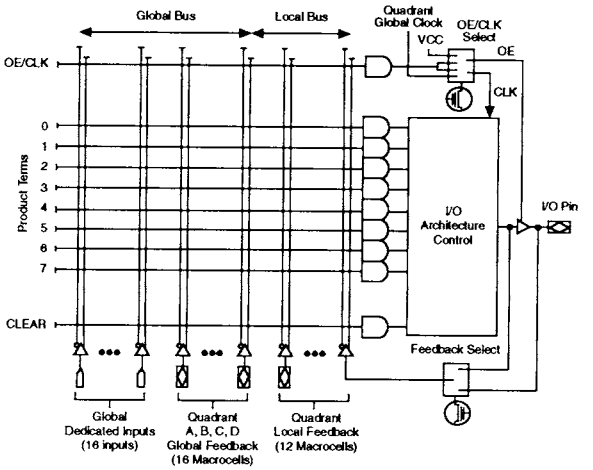


Figure 4.3.1.2 EP1810 Local Macrocell

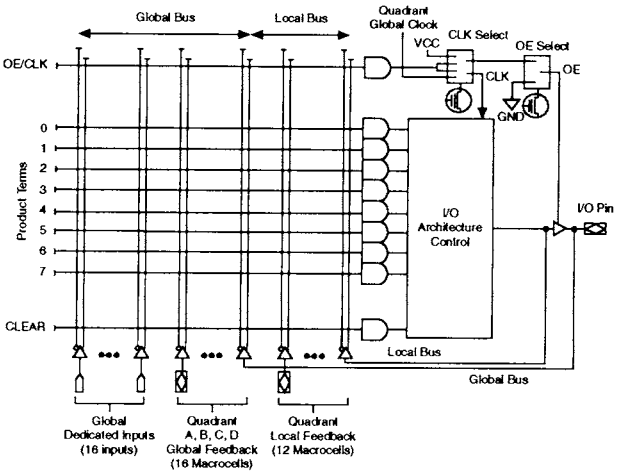


Figure 4.3.1.3 EP1810 Global Macrocell

implement buried logic functions while the associated I/O pin is used as an input. Dual feedback ensures maximum I/O flexibility.

Each macrocell consists of a logic array, a tri-state I/O buffer, and a selectable register element that can be programmed for D, T, J/K, or S/R operation, or bypassed for combinational functions. Each macrocell also has programmable output polarity. The logic array has a sum-of-products (AND/OR) structure. The 88 inputs to the programmable-AND array come from true and complement signals of the 16

dedicated data inputs, the 12 local feedback signals, and the 16 global feedback signals. The EP1810 EPLD has a total of 480 product terms distributed among 48 macrocells. Each product term represents an 88-input AND gate.

Clock Options

Each internal flip-flop in EP1810 EPLDs can be clocked independently or in user-defined groups. Each internal register may select its clock source from a dedicated global clock pin or a product term within the macrocell. Any input or internal logic function can be used as a clock.

Four dedicated global clocks (CLK1 to CLK4) also provide global clock signals to the flip-flops. One global clock is located in each quadrant; each of which is connected directly to an EP1810 external pin. Global clocks provide clock-to-output delay times that are faster than internally generated clock signals. Array clocks provide individual clocking on a macrocell-by-macrocell basis, either directly from pins or through internal logic. Array clock signals allow flip-flops to be configured for positive- or negative-edge-triggered operation. When global clocks are used, the flip-flops are triggered by the positive edge, i.e., data transitions occur on the rising edge of the clock.

Segmentation

The EP1810 was constructed using 4 quadrants, each containing 12 macrocells. These "clusters" of macrocells are called Logic Array Blocks, or LABs.

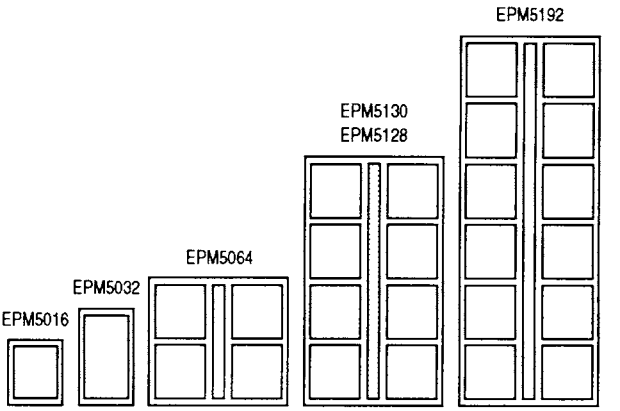
N^2 growth in the EP1810 was curtailed by going to a four quadrant (LAB) architecture. This arrangement provided local feedback to each LAB from the 12 local macrocells and global connectivity to the other LABs from 4 of the 12 macrocells in each LAB.

This architectural concept is effective because logic can be partitioned into functions whose internal connectivity is high but whose external connectivity to other functional units is limited. The intra-connectivity of signals in the LAB is very rich (every macrocell output feeds every macrocell within the LAB), while the inter-connectivity between LABs is more limited (restricted to signals generated by the global macrocells). The concept of the LAB for local functions and global lines which allow communication between modules was significantly improved in the MAX 5000 series which will be discussed next.

4.3.2 - The MAX (Multiple Array matriX) Product Family

General Description

The MAX 5000 family of EPLDs represent a significant step in the architectural evolution of programmable logic. Improvements in architecture, process technology and design result in significant increases in logic density, flexibility, and speed. The largest member of the family, the EPM5192, replaces over 100 7400-series SSI and



Macrocells	16	32	64	128	192
Maximum Flip-Flops	21	42	84	168	252
Maximum Latches	32	64	128	256	384
Pins	20	28	44	100/84/68	100/84

Figure 4.3.2.1 MAX 5000 Family

MSI packages, allowing integration of complete subsystems into a single package, saving board area, and reducing power consumption.

The MAX 5000 EPLDs range in density from 16 to 192 macrocells (see Figure 4.3.2.1). They are divided into two groups: high-speed EPLDs (EPM5016 and EPM5032) and high-density EPLDs (EPM5064, EPM5128, EPM5130, and EPM5192). The high-speed devices achieve system clock frequencies of 66 MHz, and are capable of attaining counter (of up to 32 bits) frequencies of 100 MHz, while the high-density parts achieve system clock frequencies of 35 MHz and counter frequencies of 62.5 MHz.

The modular architecture of MAX 5000 EPLDs provides integration solutions over a wide range of logic densities. Because of the uniformity of the architecture, migration from one type of device to another is easy. For example, the EPM5128 and EPM5130 EPLDs have the same logic capacity, but have packages optimized to handle different I/O requirements. The EPM5128 comes in a 68-lead package, has eight dedicated inputs and 52 I/O pins, while the EPM5130 comes in a 100-lead package and has 20 dedicated inputs and 64 I/O pins. Over the entire family, a wide range of packaging options for both through-hole and surface-mount applications is available.

Logic Array Block

The EPM5016 and EPM5032 EPLDs have a single Logic Array Block (LAB). The EPM5064, EPM5128, EPM5130, and EPM5192 EPLDs contain multiple LABs. Each LAB contains a macrocell array, an expander product term array, and a

decoupled I/O block (Figure 4.3.2.2). Expander product terms (expanders) are unallocated, inverted product terms that can be used and shared by all macrocells in the LAB. In the higher-density devices (EPM5064 and larger), macrocell output signals are routed between multiple LABs by a Programmable Interconnect Array (PIA) that ensures 100% routability. This multiple array architecture enables MAX 5000 EPLDs to offer the speed of smaller arrays (fMAX up to 62.5 MHz within the LAB of a EPM5192) with the integration density of larger arrays.

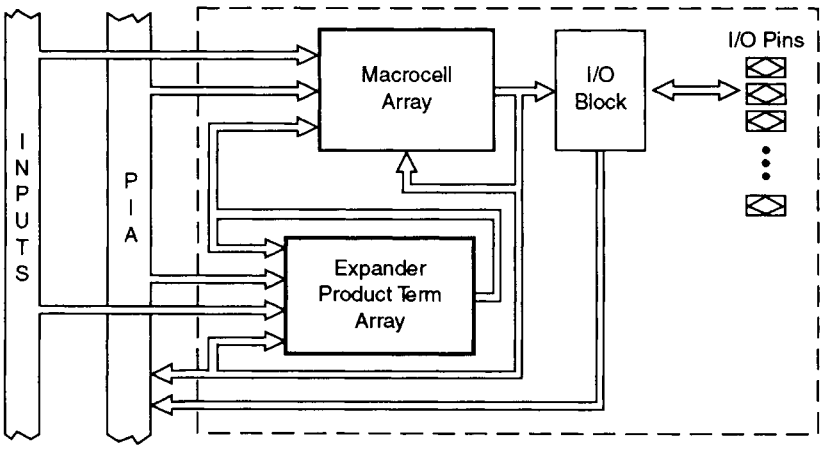


Figure 4.3.2.2 MAX 5000 Logic Array Block

MAX 5000 EPLDs contain from 1 to 12 Logic Array Blocks (LABs). Macrocells are the primary resource for logic implementation, but if needed, expanders can be used to supplement the capabilities of any macrocell. Flexible macrocells and allocatable expanders facilitate variable product term designs without the waste associated with fixed product term architectures. The outputs of the macrocells feed the decoupled I/O block, which consists of a group of programmable tri-state buffers and I/O pins. In the EPM5064, EPM5128, EPM5130, and EPM5192 EPLDs, multiple LABs are connected by a Programmable Interconnect Array (PIA). All macrocell outputs are globally routed within a LAB and also feed the PIA to provide efficient routing of signal-intensive designs.

Macrocells

The MAX 5000 macrocell, shown in Figure 4.3.2.3, consists of a programmable logic array and an independently configurable register. This register may be programmed for D, T, JK, or SR operation; or as a flow-through latch; or bypassed for purely combinational operation. Combinational logic is implemented in the programmable logic array, which consists of three product terms ORed together that feed one input of an XOR gate. The second input to the XOR gate is also controlled by a product

term. The output of the XOR gate feeds the programmable register. Expanders can be allocated to enhance the capability of the logic array.

Additional product terms, called secondary product terms, are used for output enable, preset, clear, and clock logic. Preset and clear product terms drive the active-low asynchronous preset and asynchronous clear inputs to the configurable flip-flop. The clock product term allows each register to have an independent clock and supports positive- and negative-edge-triggered operation. Macrocells that drive an output pin may use the output enable product term to control the active-high tri-state buffer in the I/O control block. These secondary product terms allow 7400-series TTL functions to be emulated exactly.

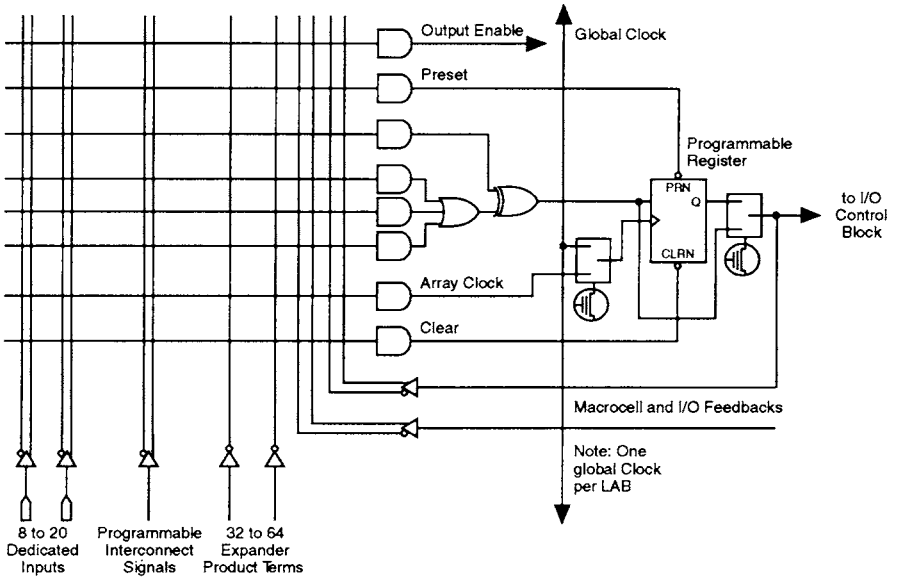


Figure 4.3.2.3 MAX 5000 Macrocell

Expander product terms

In programmable AND, fixed OR structures the chip architect must decide how many product terms is "enough" -- a typical choice has been eight. In some architectures [Kitson 1984] up to 16 product terms feed some OR gates. In most applications, eight product terms are more than enough, but are insufficient in some particular cases. This is consistent with the data published by Munoz [1987] which indicates that approximately 70% of all combinatorial functions require three (or less) product terms. Further, a substantial minority of functions require more than eight with the tail of the distribution exceeding 16.

This presents a significant dilemma for the chip architect. The choice of a high

number of product terms per OR (e.g., eight or higher) will result in a larger, slower chip where many of the resources will typically be wasted. The choice of a low number of product terms per OR (e.g., three) can result in difficulty implementing more complex functions.

The MAX 5000 architecture has three product terms per OR gate, which then feeds one input of an XOR gate, plus a fourth product term which feeds the other input of the XOR gate.

This allows implementation of all three product term functions and many four product term functions in a single pass through the array. Expanders were added to accommodate more complex logic expressions. When expanders are used the architecture becomes more like a programmable-AND, programmable-OR structure, allowing implementation of the most complex functions. However, an additional delay (the expander delay) is incurred.

The expander product term array (Figure 4.3.2.4) can be used and shared by all product terms in the LAB. Wherever extra logic is needed (including register control functions), expanders can be used to implement the logic. These expanders provide the flexibility to implement both register and product-term-intensive designs.

Expanders are fed by all signals in the LAB. One expander may feed all macrocells in the LAB or multiple product terms in the same macrocell. Since expanders also feed the secondary product terms of each macrocell, complex register control and output enable logic functions can be implemented without using additional macrocells. Expanders can also be cross-coupled to build additional flip-flops or latches [Altera AB76 1990].

Clock Options

Each LAB has two clocking modes: array and global. If array mode is chosen, each flip-flop is clocked by a product term. Thus, any input or internal logic function may be used as a clock. This allows systems that require multiple clocks to be easily integrated.

Global clocking is provided by a dedicated clock signal (CLK) from a single device pin. This direct connection provides shorter clock-to-output delay times. Each LAB has one global clock line which can be connected (programmably) to the global clock signal. If connected, all flip-flops within the LAB are positive-edge-triggered from the CLK pin. If the global clock is not connected to a particular LAB, then the flip-flops within the LAB are clocked by the individual array clock product terms associated with each flip-flop. If the CLK pin is not used as a global clock, it may be used as a dedicated input.

I/O Control Block

Each LAB has an I/O control block (Figure 4.3.2.5) that consists of a user-configurable I/O control function for each I/O pin. The I/O control block is fed by the

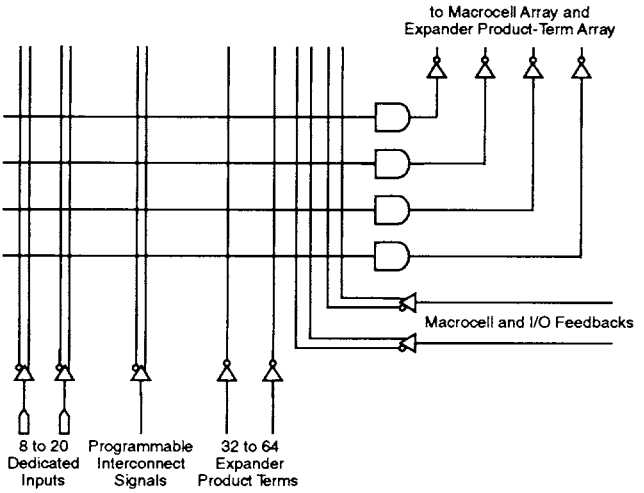


Figure 4.3.2.4 MAX 5000 Expander Product Terms

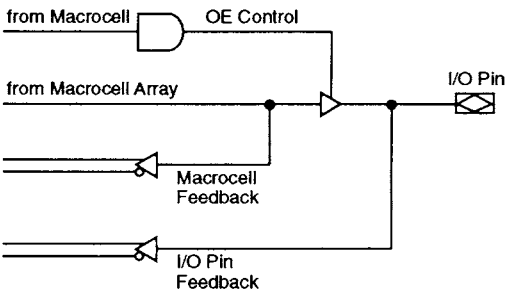


Figure 4.3.2.5 MAX 5000 I/O Control Block

macrocell array. The tri-state buffer is controlled by a dedicated macrocell product term, and drives the I/O pad.

Dual feedback -- a feedback path both before and after the tri-state buffer -- is employed for every I/O pin. This structure effectively decouples the I/O pins from the macrocells so that all macrocells within the LAB can be "buried." Thus, I/O pins can be configured as dedicated input, output, or bidirectional pins. In multi-LAB devices, I/O pins feed the PIA.

Programmable Interconnect Array

The higher-density MAX 5000 devices (EPM5064, EPM5128, EPM5130, and

EPM5192) use a Programmable Interconnect Array (PIA) to route signals between the various LABs. The PIA in the MAX 5000 devices is a fully populated cross-point switch. All I/O inputs and all macrocell outputs are inputs to the switch. There are 24 output lines per LAB. Switch connections are made by programming EPLD bits at the intersection of each unwanted crosspoint. With this PIA structure, any macrocell output signal, or any I/O input signal, may be routed to any other macrocell input without the chance of being blocked. The only routing limitation is the 24 signal fan-in limitation. The PIA has a fixed, path independent, delay, which makes timing performance easy to predict.

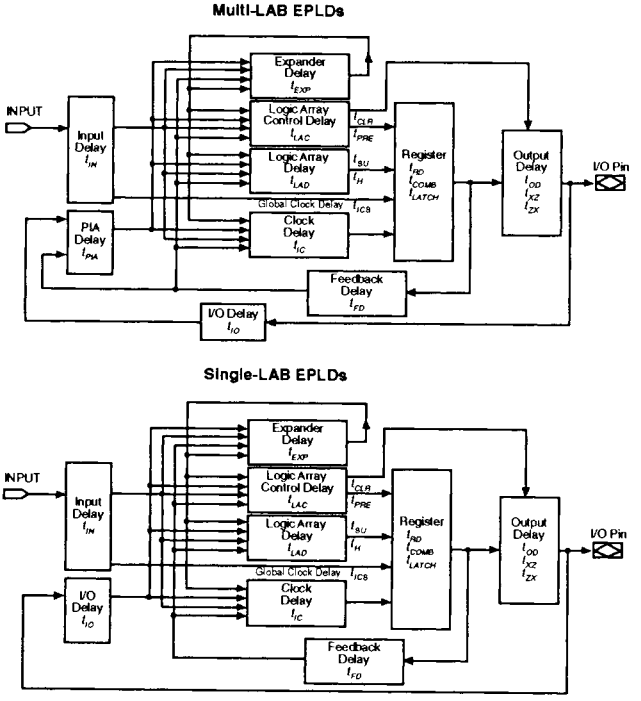


Figure 4.3.2.6 MAX 5000 Timing Model

Timing Model

Timing within multi-array parts can be determined in software, either with a Static Timing Analyzer tool, or with models as shown in Figure 4.3.2.6. Fixed internal delays allow the user to determine the worst-case timing delays for any design.

The timing models shown in Figure 4.3.2.6 may be used together with the internal timing parameters for a particular EPLD to derive timing information. External timing parameters are derived from a sum of internal parameters and represent pin-to-pin timing delays. Figure 4.3.2.7 shows the internal timing waveforms for these

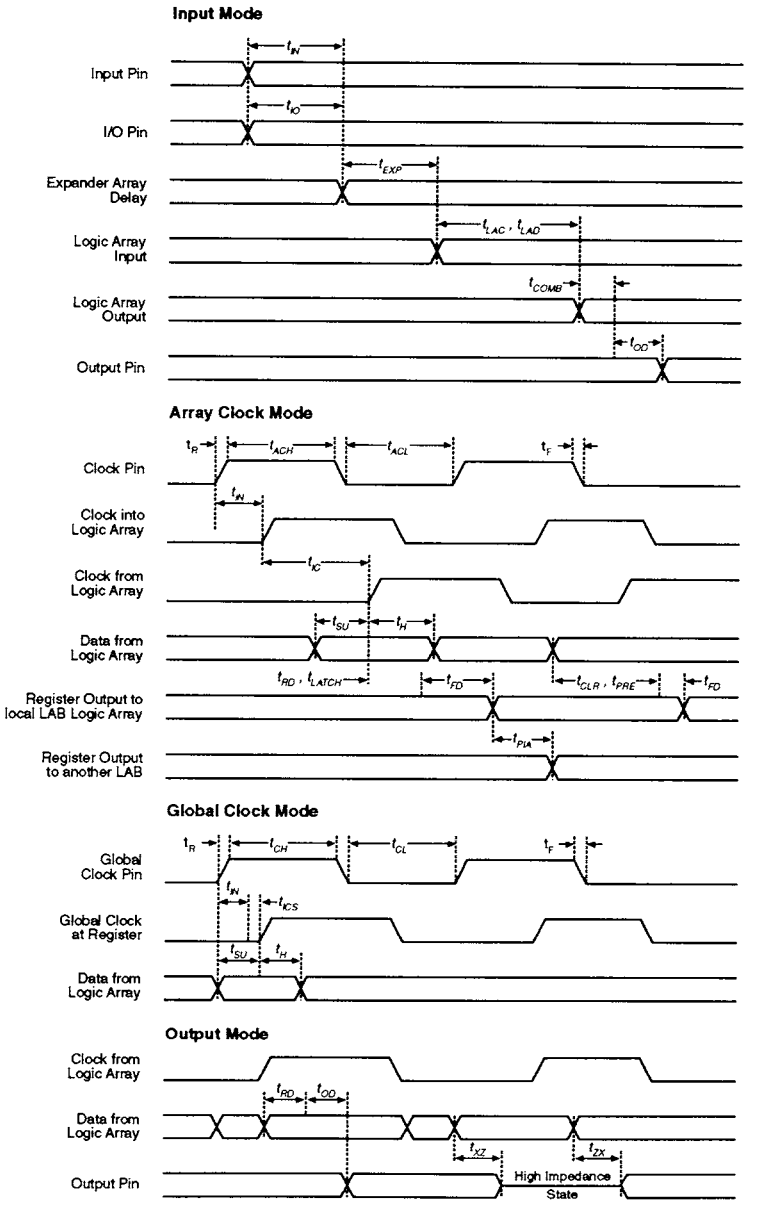


Figure 4.3.2.7 MAX 5000 Switching Waveforms

devices. Refer to Section 4.6 for further information.

4.3.3 - MAX 7000

General Description

The MAX 7000 family of EPLDs is a third generation architecture from Altera. The major goals of the MAX 7000 family are higher speed, lower power, lower cost, greater logic capability and more I/O for a given logic density. EEPROM technology is employed as the programmable element, allowing reprogrammability in non-windowed plastic packages.

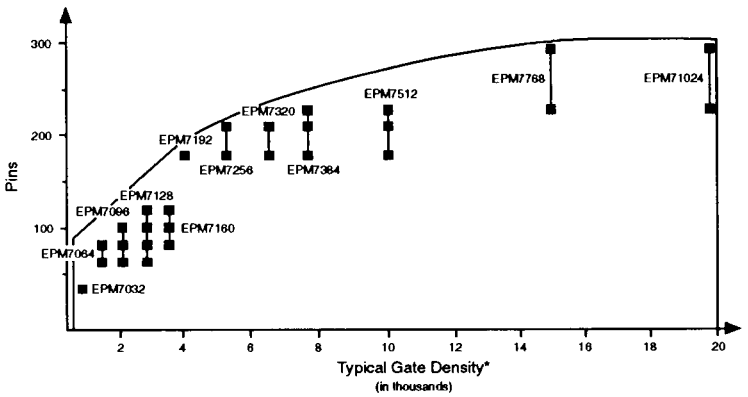


Figure 4.3.3.1 MAX 7000 Pin Count Versus Gate Density

MAX 7000 EPLDs use a multi-array style architecture to build devices with logic densities up to 10,000 gates. This family of EPLDs (shown in Figure 4.3.3.1) supports 10-ns pin-to-pin logic delays and 100-MHz system clock frequencies. Devices are housed in J-lead chip carrier (JLCC and PLCC), pin-grid array (PGA), and quad flat pack (QFP) packages, providing 44 to 288 pins.

MAX 7000 EPLDs are the first programmable logic devices with programmable speed/power optimization. Speed-critical sections of the design can run at high speed, requiring full power, while the non-speed critical sections can run at reduced speed while dissipating less than one-half of the power of full-speed operation.

The multiple-array architectural concept, pioneered in the MAX 5000 family, has been studied extensively at Altera. For example, LABs of 4, 8, 16 and 32 macrocells, both with and without expanders, have been considered. Fan-in to the LABs from the PIA was also varied. Each architecture variant was tested using a data base of over 100 design files and evaluated for utility (how many designs fit), speed, and cost (die size).

The MAX 7000 devices build on the information learned from this research. While the basic architectural choices such as LAB logic capability (e.g., 16 macrocells); global routing within the LAB; and a programmable interconnect array are carried over to the MAX 7000 family, some significant changes have been made. The key architectural changes are described below.

- A re-vamped PIA structure which provides significantly faster routing in the PIA.
- Product term selection matrix in the LAB to more efficiently use the product term resources.
- Ability to accommodate wide input OR functions (called parallel expanders) by borrowing unused product terms from adjacent macrocells.
- Programmable power saver mode. Each macrocell can be individually programmed to dissipate less power if lower speed performance is acceptable.
- Substantial increase in pin-to-logic ratio in order to accommodate I/O-intensive data path applications and 32-bit microprocessor support logic.

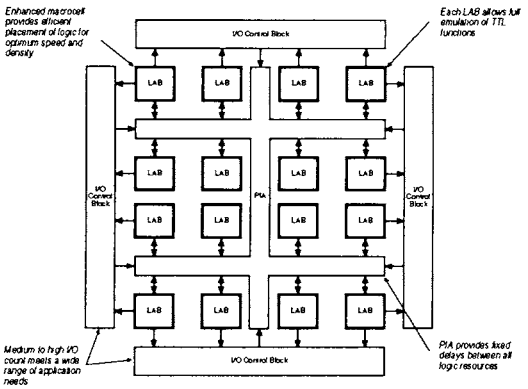


Figure 4.3.3.2 MAX 7000 Block Diagram

Functional Description

The MAX 7000 architecture (shown in Fig. 4.3.3.2) includes the following five basic elements:

- Logic Array Blocks
- Macrocells
- Logic expanders (shared and parallel)
- Enhanced programmable interconnect array
- I/O control blocks

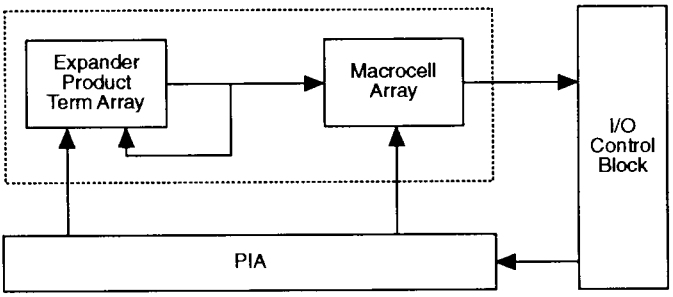


Figure 4.3.3.3 MAX 7000 Logic Array Block

Logic Array Blocks

The MAX 7000 Logic Array Block is shown in Figure 4.3.3.3. Each LAB contains 16 macrocells, and up to 16 shared logic expanders. Shared expanders can provide additional logic resources to any of the macrocells in an LAB. Furthermore, since all macrocells within an LAB share logic inputs, if one macrocell uses a specific logic input or a shared logic expander, it is also available to all other macrocells within that LAB.

Each LAB is fed by 36 inputs from the PIA, providing sufficient fan-in for the 16 macrocells to implement a wide range of typical logic functions. If more inputs are needed (for example in very wide data paths) several LABs can be used in parallel.

Macrocells

Macrocells within the LAB provide both sequential and combinational logic capability, thus ensuring the most efficient implementation of a wide range of logic functions. The MAX 7000 macrocell is shown in Figure 4.3.3.4.

Each macrocell has one flip-flop that can be programmed for D, T, JK, or SR operation with programmable clock control, individually configured for each macrocell. If necessary, the flip-flops can be bypassed for combinational operation.

Along with a programmable flip-flop, each macrocell also contains five basic product terms. These product terms can be allocated by a Product-Term-Select Matrix as primary inputs for combinational functions; as secondary inputs for individual Clear, Preset, Clock, and Clock Enable logic functions for the flip-flops; or as logic expanders to assist the generation of complex logic functions.

In addition, global Clock, Clear, and Output Enable control signals come in directly from device pins, eliminating the logic array delay and minimizing control-function delays.

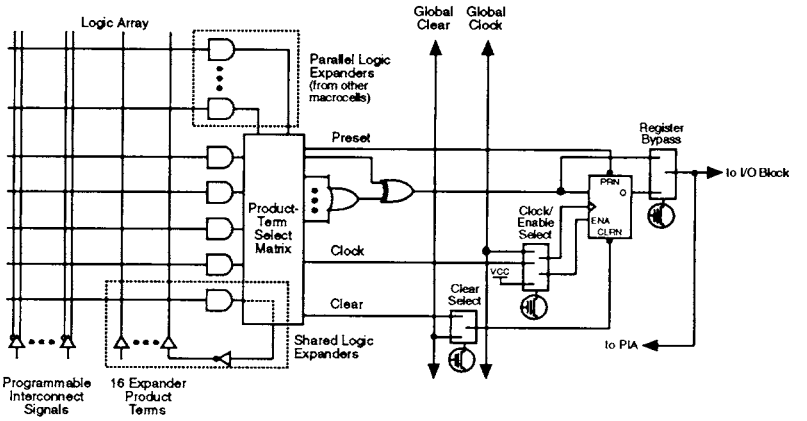


Figure 4.3.3.4 MAX 7000 Macrocell

The Clock Enable function allows flip-flops to be controlled by the logic array, even when they are clocked from the fast global Clock. This feature facilitates the implementation of high-speed synchronous designs. The Clock Enable function also allows each macrocell register to be clocked individually.

Logic Expanders

Whereas most logic can be implemented with the five basic product terms in each macrocell, some logic functions are more complex and require more product terms. Instead of using another macrocell to supply the additional logic resources, expanders are available to provide additional product terms directly to any macrocell. Unlike MAX 5000 EPLDs, which have only shared expanders, MAX 7000 EPLDs have both shared logic expanders and parallel logic expanders (see Figure 4.3.3.4).

The 16 shared logic expanders in each LAB can be viewed as a pool of uncommitted single product terms with inverting outputs that feed back into the LAB. Use of shared logic expanders enables PLA-like flexibility by allowing each shared logic expander output to be shared across all the macrocells in an LAB. Shared logic expanders can also be used to build additional register functions such as input latches.

Parallel logic expanders, on the other hand, utilize unused product terms which may be borrowed from one macrocell and assigned to an adjacent macrocell in the LAB to construct fast, complex logic. Parallel logic expanders are connected by the product-term-select-matrix in parallel with the five basic product terms in the borrowing macrocell. An additional delay of 1 to 3 nsec is incurred for this more complex logic function.

The ability to allocate additional product terms to any macrocell means that logic can be synthesized with the fewest logic resources at the fastest possible speed.

Programmable Interconnect Array

The MAX 7000 enhanced PIA is a programmable wiring path between I/O pins and LABs and from one LAB to other LABs. The PIA allows any I/O or LAB signal source to reach any destination on the device. Although it is fed by all macrocell and I/O pin feedbacks, this fast, low-skew PIA routes only those signals required to implement logic in each LAB.

The MAX 7000 PIA, shown in Figure 4.3.3.5, introduces a very short, uniform logic delay (less than 3 nsec) into the logic signal path. This has been achieved through the use of a series of two-input AND gates that feed an OR function. An EPLD bit controls one input of the AND gate and regulates the selection of the PIA signal to the LAB.

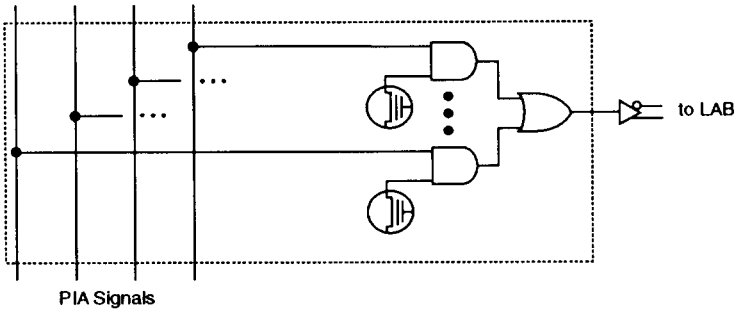


Figure 4.3.3.5 MAX 7000 Programmable Interconnect Array

I/O Control Blocks

As with other Altera architectures discussed thus far, the MAX 7000 architecture decouples I/O pins and macrocell logic resources. If an I/O pin is used as a dedicated input, the macrocell can still be used for buried logic because an independent feedback path into the logic array is provided for both the I/O pin and the macrocell.

MAX 7000 EPLDs provide two dedicated pins which can be used as global Output Enable signals. The choice of using a global Output Enable from a pin (as opposed to using a product term as was done on the MAX 5000 family) was to get maximum speed. Either OE pin can control the enable function of any output driver. The output driver can also be permanently enabled or disabled, in which case neither of the global OE signals is used. The OE input pins can also be used as general purpose logic inputs.

Routing: MAX vs. FPGAs

The ability to interconnect all points in the MAX 7000 architecture via the PIA ensures rapid, automatic design completion. Typical MAX 7000 designs can be

automatically routed in minutes. Furthermore, MAX 7000 EPLDs provide a single, short uniform delay between any signal source and all signal destinations. In contrast, some programmable gate array architectures require significant manual intervention and can take hours to route. Furthermore, incremental, additive delays between various points can cause skew and glitch problems making additional iterations of the design necessary.

Programmable Speed/Power Control

The MAX 7000 family offers a programmable speed/power tradeoff that supports reduced-power operation across selected signal paths or the entire device. Other PLDs (including PALs and some members of the Altera Classic family) offer half-power or quarter-power operation across the entire device usually at the cost of reduced speed across the entire device. In the MAX 7000 family, each macrocell can be programmed by the designer for either high-speed or low-power operation. The small (less than 5 ns) speed penalty for (approximately) quarter-power operation applies only to those macrocells selected for low power. As a result, speed-critical portions of the design can run at high speed while the remainder of the design can operate at lower-power. Since only a small fraction of all gates operate at maximum frequency in most logic applications, this feature allows typical power savings of up to 50% when compared to standard PLD implementations.

4.3.4 - MPLDs: Mask-Programmed Logic Devices

General Description

Mask-Programmed Logic Devices (MPLDs) provide a masked alternative to EPLD designs. By using a generic CMOS process and removing all EPROM cells, considerable die cost savings can be achieved. MPLDs are appropriate for designs which are no longer likely to change and for which high volume is anticipated. A combination of EPLDs for prototyping and production ramp-up and MPLDs for high volume production provide both fast time-to-market and low cost and low risk. See Figure 4.3.4.1.

The EPLD-to-MPLD conversion requires no redesign effort. The MPLD is guaranteed to meet the worst-case AC and DC parameters of the original EPLD design. Test vectors are automatically generated as part of the conversion process.

EPLD/MPLD Compatibility

A MPLD is guaranteed to be pin-, function-, and timing-compatible with the original EPLD design. This guarantee ensures that the MPLD can replace the EPLD without interrupting production of the end system.

Pin compatibility guarantees that both the pin-out and DC specifications of the MPLD match those of the original EPLD design. In addition, the MPLD typically will consume less than one-tenth of the power of the equivalent EPLD, depending on the

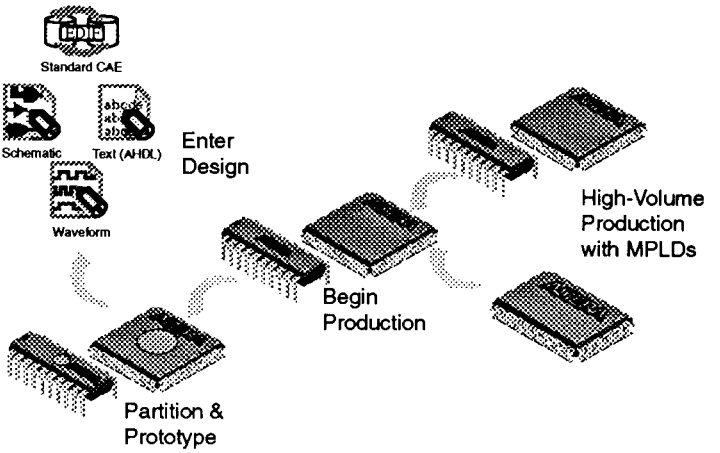


Figure 4.3.4.1 EPLD/MPLD Development Flow

design and operating conditions.

Functional compatibility of the MPLD is ensured by directly mapping the primitives within the EPLD (product terms, programmable flip-flops, etc.) to specially designed elements within the MPLD. A proprietary logic synthesis program that uses the Simulator Netlist File (SNF) generated by MAX+PLUS II software is employed in the conversion process. The SNF reflects the final synthesis, placement, routing and signal timing of the original EPLD design. The conversion process pays special attention to the wide fan-in product terms and the wide fan-out of macrocells commonly found in EPLD applications.

MPLDs are guaranteed to meet the worst-case data sheet timing parameters of the corresponding EPLD. If the design engineer performs worst-case analysis of the EPLD, the same analysis will hold for the MPLD. Therefore, the timing of the original design and the overall system is maintained when the EPLD is replaced with an MPLD.

Design for Testability

Test vector generation is one of the most time-consuming tasks required for ASIC design. A significant advantage of EPLDs over ASICs is that they are fully tested before they are shipped. The pattern programmed into the device is also verified at programming time. Application specific test vectors are not required.

MPLD designs include a partial scan-testing structure that parallels the testability available in EPLDs. The partial-scan structure allows creation of test vectors with over 95% fault coverage for all stuck-at and open faults. This high fault coverage is maintained regardless of whether synchronous or asynchronous design techniques are

used. [Ahanin 1992]

The built-in design-for-testability frees the design engineer from the burden of creating a testable design and test vectors. In addition, customer-provided simulation vectors are optional and can be appended to the test vectors generated during the conversion process.

N-to-1 Conversion Option

Many applications use multiple EPLDs on a single board for both prototyping and production. In some applications it may be desirable to perform prototyping with multiple EPLDs, and then convert the design to a single-device for high-volume production (see Figure 4.3.4.1). The EPLD-to-MPLD conversion program provides this capability with the "N-to-1" conversion option, which offers the benefits of developing with EPLDs even when production constraints require a high-density single-device solution.

The N-to-1 option allows a multi-EPLD design to be converted into a single MPLD that is function- and timing-compatible with the original multi-EPLD solution. The package and pin-out is determined by the application's requirements. A wide range of package options is available.

Quick, Seamless Conversion

One of the principal objectives of the EPLD-to-MPLD conversion program is to minimize the design engineer's involvement in the conversion. With an automated conversion process, the engineer is free to begin developing the next-generation project.

The MPLD design flow chart (see Figure 4.3.4.2) shows how an EPLD is converted to an MPLD. The design engineer submits a "design packet" consisting of design files and design checklists from which a quotation is generated. Next, an engineer reviews the design and submits a Final Design Sign-off Form for customer approval. This form describes the specifications of the MPLD in detail. Following customer approval, the design conversion takes place.

The design conversion includes netlist translation, logic synthesis, testability insertion, Automatic Test Vector Generation (ATVG), fault grading, timing analysis, place-and-route, post-route timing analysis, design validation, and the manufacture of the prototypes. The entire conversion, from final design sign-off to prototype delivery, takes less than 5 weeks (6 weeks for N-to-1 conversion). Production quantities are delivered 10 to 12 weeks after the customer returns the Prototype Sign-off Form.

Summary

Two important design goals faced by engineers today are reducing time-to-market and system cost. Figure 4.3.4.3 illustrates that a combination of EPLDs and MPLDs can provide a solution that achieves these goals.

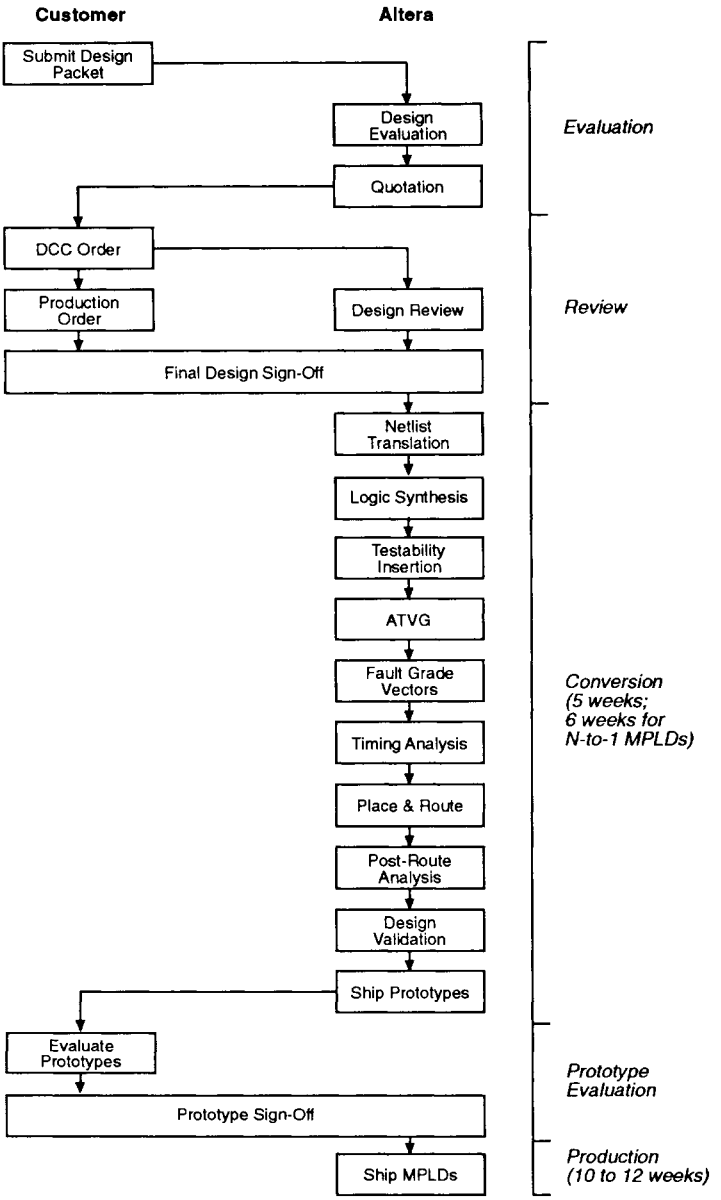


Figure 4.3.4.2 MPLD Design Flow Chart

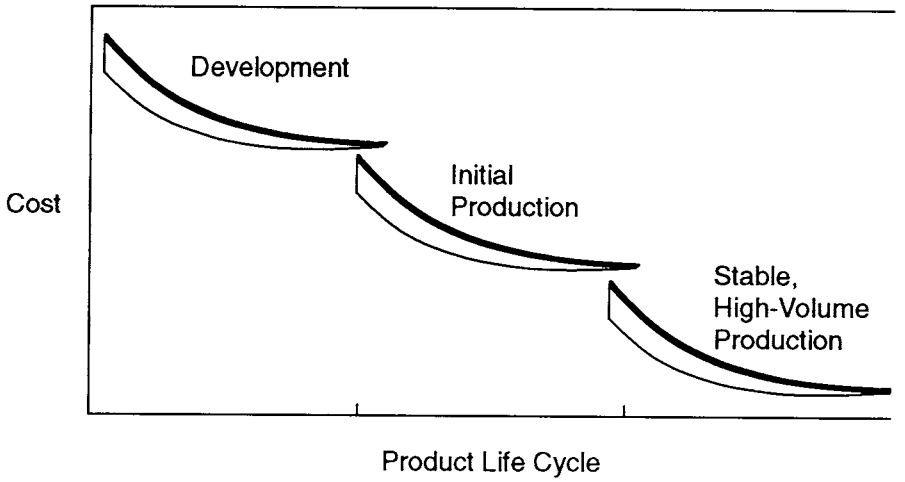


Figure 4.3.4.3 EPLD/MPLD Cost Versus Volume

4.4. Software

In previous sections the architectures of three EPLD product families were discussed in some detail. However, the philosophy of most PLD suppliers is that a logic designer need not understand the inner complexities of EPLD architectures. The user may work with familiar design entry tools (e.g., TTL functions or a high-level design language), and software should automatically translate design intent into the format required to fit the chosen EPLD architecture.

For optimum results, software products are developed together with and impose a significant influence on the EPLD architectures. Software tools that support familiar design entry methods and rapid design completion are the desired result. In this environment a user can take a logic circuit from design entry to device programming in a matter of hours (see Fig. 4.4.1). Design processing is typically completed in minutes, allowing many design iterations to be completed in a single day.

Typically, software is available for X86 PCs and Sun, HP, DEC and other workstation computers. Many design entry options are available: hierarchical schematic capture (with basic gate and complete TTL libraries), the hardware description language (AHDL, VHDL and Verilog), Boolean equation, truth table, netlist, and waveform entry. (See Figure 10.2) Design entry methods may be freely combined to create a single EPLD design. Design compilers perform minimization and logic synthesis, design fitting (analogous to automatic place-and-route), and generate programming data. Design verification via functional simulation, timing simulation, and delay prediction for speed-critical paths is also available. Hardware for programming EPLDs is offered by Altera and many PLD programmer manufacturers. Software

interfaces to other design tools are provided by translators, and via industry-standard EDIF [ANSI 1988] [Datta 1991] netlists. Many third-party compilers also support EPLDs directly.

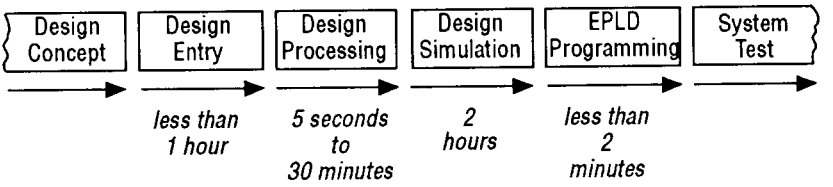


Figure 4.4.1 EPLD Design Methodology

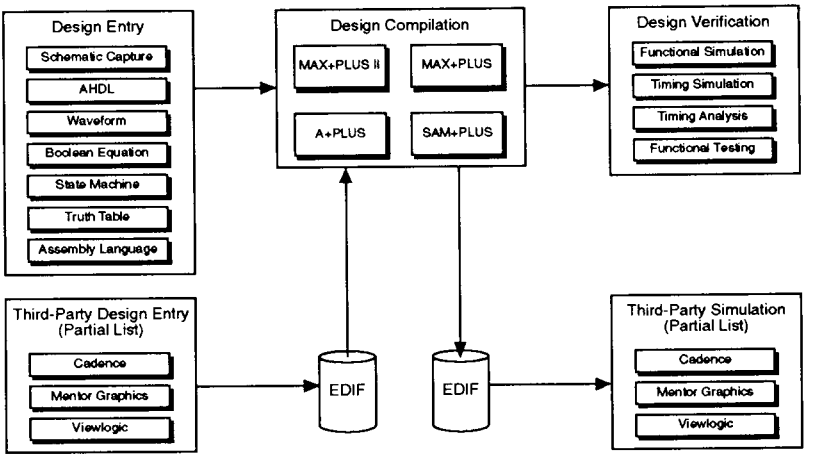


Figure 4.4.2 Altera Design Environment

MAX + PLUS II

Altera's current generation of PC-based CAE tools, MAX+PLUS II, runs in the Windows environment on PCs and in the Motif environment on workstations. The key features of MAX+PLUS II are listed below.

MAX+PLUS II Features

- Single, unified development system providing support for Altera's Classic, MAX 5000 and MAX 7000 EPLDs.
- Runs under Microsoft Windows.
  - intuitive graphical user interface
  - efficient memory management

  - multi-tasking capability
  - extensive printer/plotter support.
- Hierarchical graphic, text, and waveform design entry:
  - Graphic Editor for schematic-based designs
  - Text Editor for high-level textual descriptions
  - Waveform Editor for design entry and editing/viewing simulation results
- Applications run concurrently, allowing multiple editors to be active simultaneously, while simulations or compilations run in the background.
- Automatic error location is provided in the Graphic, Text, and Waveform Editors.
- Partitioning automatically divides large designs into multiple EPLDs.
- VHDL, Verilog and AHDL support.
- Logic synthesis and minimization.
- Functional simulation for detailed functional debugging.
- Interactive timing simulator supports multi-EPLD designs and probes for viewing internal nodes.
- Bidirectional Electronic Design Interchange Format (EDIF 2 0 0) netlist interface.
- On-line, context-sensitive help.

General Description

MAX+PLUS II includes design entry, compilation, multi-chip partitioning, timing simulation, and device programming support. Figure 4.4.3 shows a block diagram of MAX+PLUS II.

MAX+PLUS II supports three hierarchical design entry mechanisms: (1) schematic designs are entered with the Graphic Editor; (2) text descriptions using AHDL or EDIF 2 0 0 netlists are entered with the Text Editor; and (3) waveforms are entered with the Waveform Editor. All editors can be used concurrently, with multiple files open at a time. A library of over 300 7400-series TTL and custom macrofunctions for both schematic and text designs is included.

The Compiler synthesizes and optimizes designs using advanced logic synthesis and minimization techniques together with heuristic fitting rules to efficiently place and route the design within an EPLD. A programming file is created by the Compiler to program the EPLDs.

The multi-device partitioning automatically splits large designs (beyond the capacity

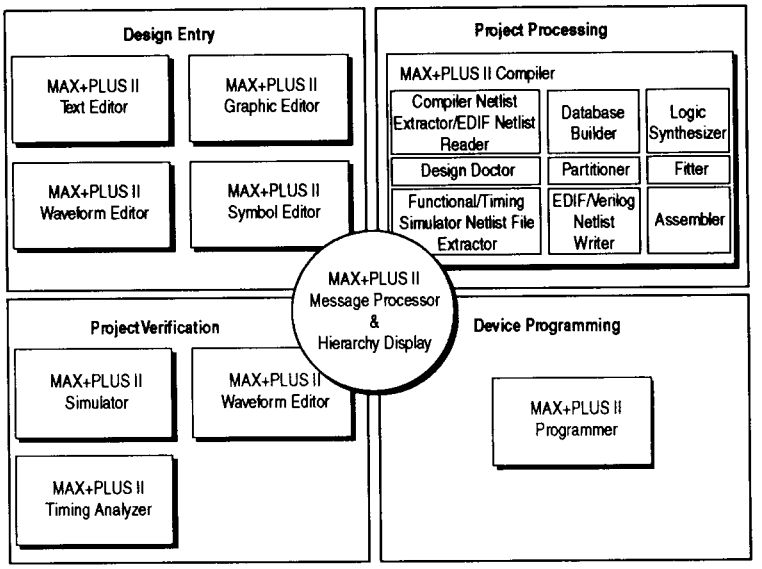


Figure 4.4.3 MAX+PLUS II Block Diagram

of a single EPLD) into multiple EPLDs, allowing the user to create large system-level designs.

The Simulator performs event-driven timing simulation. It supports multi-EPLD simulations and interactively displays timing results in the Waveform Editor. With the Waveform Editor, the user can enter, modify, and group input vectors; view simulation errors; and compare simulation runs.

The Compiler reports any design errors to the Message Processor, which automatically highlights the source of an error in the Graphic, Text, or Waveform Editor. MAX+PLUS II is fully integrated with the Windows Clipboard. The designer uses the Clipboard to quickly copy design information from one editor to another, while extensive on-line help provides instant information on all aspects of the system. The Hierarchy Display lets the designer move between hierarchical design files by simply selecting an icon.

Design Entry

Design entry files - graphic, text, and waveform - can be mixed freely. In addition to VHDL, Verilog and AHDL Text Design Files, MAX+PLUS II also accepts EDIF 2 0 0 netlists produced by CAE tools from vendors such as Cadence, Data I/O, Mentor Graphics, OrCAD, Synopsys and Viewlogic.

The Graphic, Symbol, Text, and Waveform Editors can open windows on several files at the same time. For example, a Waveform Editor window can display simulation

results, while the Text Editor shows an AHDL description. At the same time, the user can open two windows of the Graphic Editor that display different levels of a design's hierarchy, or even show different areas of the same design file. If one design is displayed in two windows of an editor, any edits made in one window are automatically reflected in the other.

Graphic & Symbol Editors

The Graphic Editor (Figure 4.4.4) provides a convenient tool for schematic design entry. Designers can enter probes into the schematic to trace a specific signal (e.g., flip-flops, logic outputs) during simulation. Tag-and-drag editing can be used to quickly move individual symbols, groups of items, or entire areas. During a move, a net can be broken or connections can be preserved with orthogonal rubberbanding. Other Graphic Editor features include the ability to group nodes into buses, locate the source and destination of nets, and make quick net name changes with the search-and-replace feature.

The Compiler automatically generates a symbol that represents a design file, which can then be used in a higher-level schematic. The designer can use the Symbol Editor to modify input and output pin placement or to customize the appearance of an automatically created symbol.

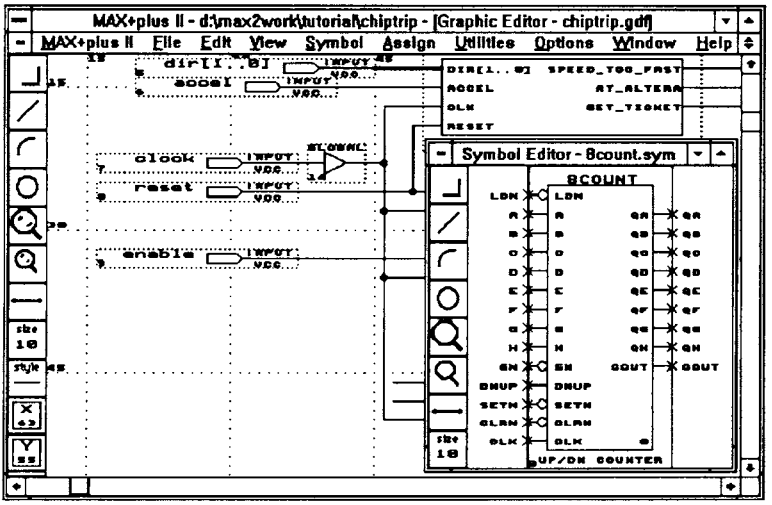


Figure 4.4.4 MAX+PLUS II Graphic Editor

Text Editor & AHDL

The Text Editor lets the user view and edit any ASCII text file in the MAX+PLUS II environment, including AHDL Text Design Files, Vector Files, Report Files, and

EDIF netlists. A Text Editor window that contains a Text Design File is shown in Figure 4.4.5. The AHDL language syntax supports arithmetic and relational operations such as addition, subtraction, equality, and magnitude comparisons. Standard Boolean functions, e.g., AND, OR, NAND, NOR, XOR, and XNOR, are also included. Since AHDL supports groups, operations can be performed on a byte- or word-wide basis as well as on single variables. AHDL also allows the designer to specify the location of nodes within Altera EPLDs. Together, these features make it easy to implement complex designs in a concise, high-level description.

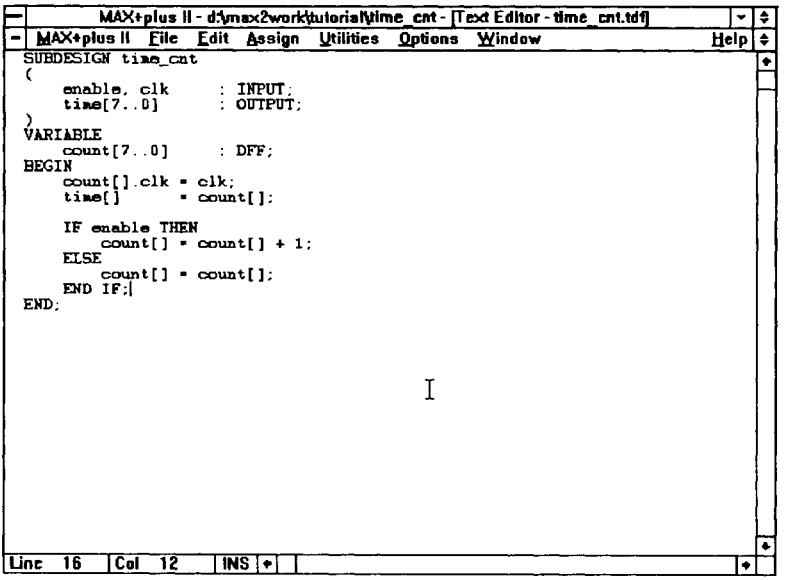


Figure 4.4.5 MAX+PLUS II AHDL Environment

Waveform Editor

The Waveform Editor (Figure 4.4.6) is used to create and edit waveform designs. In addition, the Waveform Editor functions as a logic analyzer that allows the designer to view and edit simulation results.

Designs that generate timing signals are best described with waveforms. The Compiler's waveform synthesis algorithms automatically generate logic from user-defined input and output waveforms.

Registered and combinational logic as well as state machines can be described with waveforms. The Compiler determines the optimal number of state bits and state variable assignments.

The Waveform Editor includes features to define and modify Waveform Design Files

and input vectors for simulation. The designer can:

- copy, cut, paste, repeat, and stretch waveforms;
- add or delete internal nodes, flip-flops, and state machines;
- combine waveforms into binary, octal, decimal, or hexadecimal buses;
- compare the differences between two simulations by simulating a design, editing the input vectors, and then simulating the design again. The Waveform Editor superimposes the output waveforms for easy comparison.

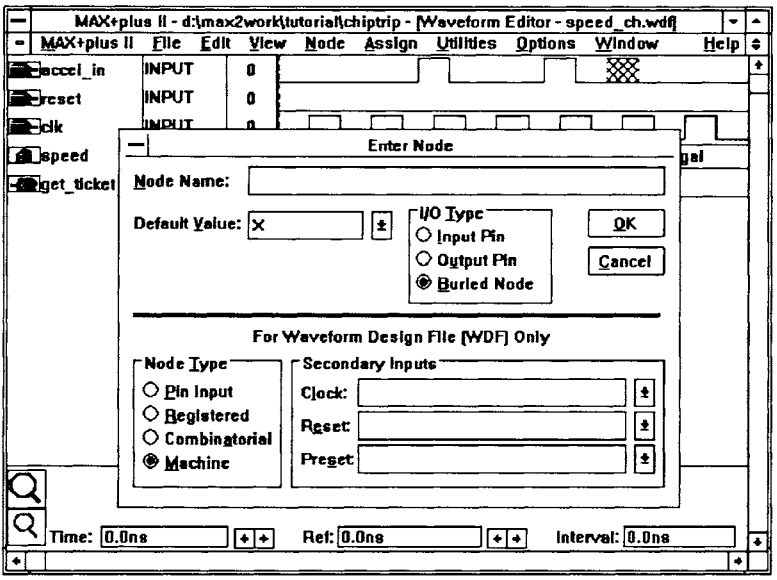


Figure 4.4.6 MAX+PLUS II Waveform Editor

Hierarchy Display

The Hierarchy Display shows the current project hierarchy - including all lower-level design files - which can be a mixture of graphic, text, and waveform files. The user selects one or more files, and MAX+PLUS II then opens the appropriate editor to display the design. This "context-sensitive editing" feature makes it easy to move around the project hierarchy.

Clipboard

The Windows 3.0 Clipboard is a temporary storage location that allows users to pass design information between editors. Text from a Text Editor file can be copied into the Graphic, Symbol, and Waveform Editors. Schematics can be copied between

Graphic Editor files, and waveforms can be pasted from one Waveform Editor file to another. Information can also be pasted into other Windows 3.0 applications.

Macrofunction Library

The MacroFunction Library contains over 300 7400-series TTL, bus, and EPLD-optimized functions. All have been optimized for speed and device utilization, and all perform true TTL emulation. Table 1 lists some of the macrofunctions currently available.

Table 1. Partial List of MAX+PLUS II Macrofunctions

Type	**Macrofunctions**
Adder	8FADD, 7480, 7482, 7483, 74183, 74283, 74385
ALU	74181, 74182, 74381, 74382
AND-OR Gate	7452
Comparator	8MCOMP, 7485, 74518, 74684, 74686, 74688
Code Converter	74184, 74185
Counter	4COUNT, 8COUNT, 16CUDSLR, GRAY4, UNICNT, 7493, 74160, 74161, 74162, 74163, 74190, 74191, 74192, 74193, 74393...
Decoder	7442, 7443, 7444, 7445, 7446, 7447, 7448, 7449, 74138, 74139, 74154, 74155, 74156...
Encoder	74147, 74148
Frequency Divider	FREQDIV, 7456, 7457
Latch	INPLTCH, NANDLTCH, NORLTCH, 7475, 7477, 74116, 74259, 74279, 74373...
Multiplier	MULT2, MULT4, MULT24, 74261...
Multiplexer	21MUX, 74151, 74153, 74157, 74158, 74298...
Parity Generator	74180, 74280
Register	7470, 7471, 7472, 7473, 7474, 7476, 7478, 74173, 74174, 74175, 74178, 74273, 74374...
Shift Register	BARRELST, 7491, 7494 ,7496, 7499, 74164, 74165, 74166, 74179, 74194, 74198...
SSI Gate	CBUF, INHB, 7400, 7402, 7404, 7408, 7410, 7411, 7420, 7421

EDIF Support

MAX+PLUS II has a built-in bidirectional EDIF 2 0 0 netlist interface, providing a convenient bridge to popular CAE schematic capture, synthesis and simulation tools. Any CAE software package that produces EDIF 2 0 0 netlists can export designs to MAX+PLUS II with Library Mapping Files (.LMF) that convert vendor CAE functions to equivalent Altera primitives and macrofunctions. Altera provides a number of ready-made LMFs for popular software packages from vendors such as Mentor Graphics, Cadence, and Viewlogic, but users can also create their own LMFs to map any CAE software library. MAX+PLUS II then automatically generates a

symbol from a translated EDIF file, so that the file can be directly incorporated into a schematic or AHDL design.

EDIF netlists can also be exported from MAX+PLUS II to third-party CAE tools, allowing the user to simulate EPLD designs on the workstation of choice. Output Mapping Files can convert Altera primitives and macrofunctions to equivalent workstation functions.

Design Processing

The MAX+PLUS II Compiler processes designs for all Altera general-purpose EPLDs, including the Classic, MAX 5000, MAX 7000, and STG EPLDs (see Figure 4.4.7).

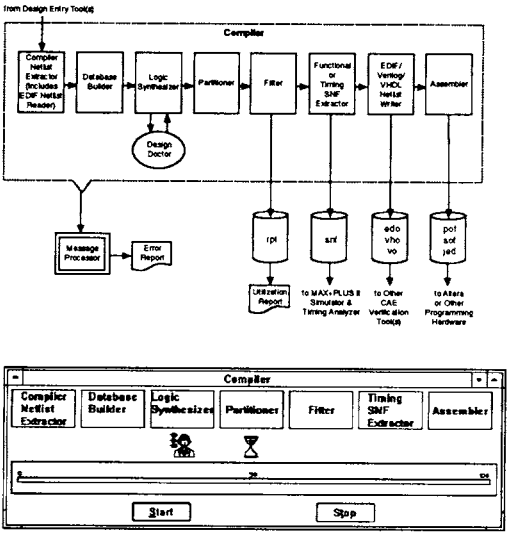


Figure 4.4.7 MAX+PLUS II Compiler

Compiler options simplify design processing and analysis. The user can specify the degree of detail of the Report File that shows how an EPLD has been utilized. The user can also specify the target EPLD family for the design, and whether or not the Compiler should extract a Simulator Netlist File for simulation within MAX+PLUS II and an EDIF Output File for third-party simulators.

The first module of the compiler, the Compiler Netlist Extractor, extracts the netlist used to define the design. This module also contains a built-in EDIF Netlist Reader. The Compiler Netlist Extractor checks design rules for any errors. If errors are found, they are passed to the Message Processor, which can then locate them in the appropriate editor. A successfully extracted design is built into a database and passed to the Logic Synthesizer.

The Logic Synthesizer module invokes algorithms that translate and optimize the user-defined logic to make the most efficient use of the resources of the target architecture. The Logic Synthesizer uses expert synthesis rules based on the target architecture (Classic, MAX 5000, MAX 7000, or STG) to factor and map logic within the chosen EPLD structure.

For simple PLD architectures, the conventional logic synthesis algorithm has been to expand the designer's logic to a sum-of-products form and then minimize the sum of products to arrive at an expression containing a minimum of product terms. If the final expression is within the product term limit for the target device, then the algorithm is successful. If not, the user must change the design and try again.

MAX-family EPLDs incorporate XOR gates and sharable, multi-level expander product terms that can be allocated wherever additional combinational logic is needed. Figure 4.3.2.3 shows a diagram of the architecture of a MAX EPLD.

The Logic Synthesizer simultaneously applies several techniques to implement design logic in the targeted device. It keeps track of the method(s) that use the fewest resources (i.e., expander product terms), and keeps only the results of the best method. The synthesis algorithm uses all combinations of techniques that are applicable in a given situation. Techniques that take advantage of the XOR gate are applied in combination with those that use the expander product terms to arrive at the best synthesis.

Eight of these logic synthesis techniques are summarized below.

1. Expansion to and Minimization of the Sum of Products

The design logic is expanded to a sum-of-products form, after which the minimization algorithm calculates the sum-of-product form with the minimum number of product terms.

2. Fitting Directly

If the logic can be expressed in a sum-of-products form without using more product terms than are available in the device, then the logic is placed directly on the product terms. Therefore, in MAX EPLDs, if the expression has three or fewer product terms, it is placed directly on the three product-term inputs of the XOR gate. (See Fig. 4.3.2.3.)

3. DeMorgan's Inversion

The Logic Synthesizer also tries both the original form and the complemented form of the design logic, in an attempt to use fewer product terms (the second input of the XOR gate is used to reinvert the logic). DeMorgan's inversion and techniques #1 and #2 above are essentially the only logic synthesis methods used in the previous generation of programmable logic compilers.

4. Fitting on Expanders

If an expression has too many product terms to fit directly, each of the product terms can be placed on an expander product term that is essentially a NAND gate. All of these expander product terms are then ANDed together on one of the product terms of the XOR gate, while the other input is used to invert the final result. (See Figures 4.3.2.3 and 4.3.2.4.) This is a straightforward NAND-NAND implementation that can always be used, no matter how many product terms there are in the sum-of-products expansion of the design logic.

5. Using XOR Gates in the Design

Instead of immediately expanding the design logic to a sum-of-products form, the Logic Synthesizer first checks whether the design uses an XOR gate as the first gate of the network of combinatorial gates. It then tries all possible ways to fit the inputs of the design's XOR gate onto the inputs of an XOR gate in a macrocell. Thus if P and Q are the inputs to the User's XOR gate, the Logic Synthesizer will try all four valid ways of placing P, Q, not P, and not Q on the "3 and 1" product term inputs of the macrocell's XOR gate (i.e., try XOR (Q, P), XOR (P, Q), XOR (not Q, not P) and XOR (not P, not Q)).

6. Factoring

The Logic Synthesizer also tries to factor a sum-of-products expression into a multi-level sum of products of sums which can be implemented on the expander product terms using NAND-AND-OR logic. The algorithm finds subsets of product terms that differ by only a single factor. These subsets are then factored into a sum of single factors ANDed with the remainder of the products. This sum is then factored out and treated as a single factor for the rest of the analysis, which continues to search for additional factorable subsets.

For example, the following sum-of-products form would naively (by technique #4) require six product terms. It can be transformed as shown below to use only two expander product terms (* is the AND operator, + is the OR operator, $ is the XOR operator, and / is the prefix inversion operator):

```
z   = a * b * c * p * x
    + a * b * c * p * y
    + a * b * c * q * x
    + a * b * c * q * y
    + a * b * c * r * x
    + a * b * c * r * y;
```

after factoring once:

```
z   = a * b * c * (x + y) * p
    + a * b * c * (x + y) * q
    + a * b * c * (x + y) * r;
```

after factoring again:

```
z   = a * b * c * (x + y) * (p + q + r);
```

and after converting to NAND for the expander product terms:

```
z   = a * b * c
    * /( /x * /y)
    * /( /p * /q * /r );
```

Thus the final expression uses only one product term and two expander product terms (the expressions in parentheses are implemented on expander product terms).

7. Finding Co-sets

This method begins with an algorithm that divides a sum-of-products expression into two subsets of product terms, such that whenever one of the subsets is true, the other is false. If two such sets are found, they are implemented on the XOR gate (when only one input is true at any given time, the XOR will function as an OR gate). The algorithm used to find two cosets of product terms starts by considering all possible pairs of product terms. If a pair of product terms has a common factor that is used in opposite polarity, then they are considered as potential seeds of the two cosets of product terms. After a pair is found, the remaining product terms are examined individually to determine whether they fit into the coset with the first or second product term. If a product term cannot go into either set, the original seed pair is discarded and another is tried until two cosets are found or until all pairs have been tried.

A 4-to-1 multiplexer provides a good example of this technique:

```
Q   = A * X * Y
    + B * X * /Y
    + C * /X * Y
    + D * /X * /Y;
```

The multiplexer fits on the XOR gate as follows (using NO expander product terms):

```
Q   = ( A * X * Y
    + B * X * /Y
    + C * /X * Y)
    $ ( D * /X * /Y);
```

8. Using the XOR as Expansion Logic

If the expression to be synthesized is the inverse of a sum of some number (N) of product terms (where N is greater than three for a MAX macrocell), this technique can always fit the logic using N-3 expander product terms. This technique relies on the following identity:

if

$$Q = /(\_A_i + \_B_j);$$

then

Q = ((_ A_i) * _ /B_j) $ (_ /B_j);

which becomes evident when it is shown that Q is true if and only if all A_i and all B_j are false in both expressions. (A_i and B_j represent product terms.)

For example, the following inverted expression (which naively would require five expander product terms by technique #4 above):

Q = /(A * B * C + D * E + F * G + H * I + J * K);

becomes a "3 and 1" product term expression using the XOR:

Q = ((A * B * C + D * E + F * G) * /X * /Y) $ (/X * /Y);

and only two expanders:

X = H * I;
Y = J * K;

For large system-level designs, the Partitioner is invoked. The Partitioner uses a sophisticated "Min-Cut" algorithm [Kernighan 1970] [Fiduccia 1982] to separate the logic design into multiple EPLDs from the same family, relieving the designer of the time-consuming task of manually splitting a large design into smaller designs. The user can control the design's partitioning by entering specific chip assignments for flip-flops and pins in the source design files.

After partitioning, the Fitter applies heuristic rules to optimally place the synthesized design into one or more chosen EPLDs. In devices with PIA structures - i.e., larger MAX 5000 and MAX 7000 EPLDs - or with local/global bus structures such as the EP1810 EPLD, the Fitter also automatically routes signals across this interconnect to relieve the designer of tedious place-and-route tasks. The Report File issued by the Fitter shows design implementations as well as any unused resources in the EPLDs.

The Simulator Netlist Extractor generates a netlist from the compiled design if the user desires simulation or timing analysis data.

The EDIF Netlist Extractor can produce an EDIF 2 0 0 netlist that contains all post-synthesis function and delay information for the completed design, so that it can be integrated into a workstation environment [Altera 1991].

Finally, the Assembler module creates one or more Programmer Object Files (POF) and/or JEDEC Files from the compiled design. The MAX+PLUS II Programmer uses these files and standard Altera hardware to program the desired EPLDs.

Message Processor

The MAX+PLUS II Message Processor is the clearinghouse for all messages generated during compilation, simulation, timing analysis, and programming. For example, if an error occurs during compilation, the Message Processor displays a brief description of the error. The user then selects the error message, chooses the

Locate button, and the troublesome logic is highlighted in the appropriate editor. Or, if a set-up time violation occurs during simulation, the Message Processor invokes not only the Waveform Editor to highlight the portion of the simulation waveform where the violation occurred, but also the appropriate editor to show the specific flip-flop location.

Functional Simulation

Functional simulation (Figure 4.4.8) allows the user to test the logical operation of a design before compilation is completed. The designer can quickly identify and correct logical errors in a design without first having to synthesize, partition, and fit the logic into an EPLD. Functional simulation is performed in the MAX+PLUS II Simulator. The Waveform Editor displays the results of functional simulation and provides easy access to all nodes in a design, including combinational functions.

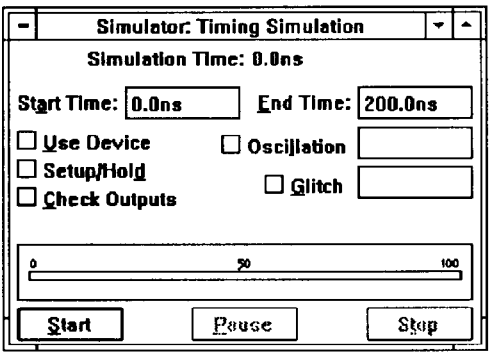


Figure 4.4.8 The MAX+PLUS II Simulator

Timing Simulation

The designer either defines input stimuli with a straightforward vector input language, or draws waveforms directly with the Waveform Editor. Simulation results can be viewed in the Waveform Editor, or printed out as table and waveform files.

The Simulator uses the Simulator Netlist File extracted from a compiled design to perform timing simulation with 0.1-ns resolution. The user can specify commands either interactively or in a batch file to perform a variety of tasks, such as halting the simulation when user-defined conditions are met or forcing flip-flops high or low.

If flip-flop setup or hold times have been violated, the Simulator warns the user, sending the information about where and when the problem occurred to the Message Processor. Also, if the user-defined minimum pulse width and period of oscillation are violated during simulation, the Message Processor locates the offending node in the original design file, and displays the time at which the problem took place in the Waveform Editor.

Differences between two simulations are viewed in the Waveform Editor, where the simulation results can be superimposed for easy comparison.

Timing Analysis

MAX+PLUS II software includes analysis tools for analyzing the timing of a completed design. The user simply tags start and end points in the Graphic, Text, or Waveform Editor to enable the Timing Analyzer to determine the shortest and longest propagation delays. The Timing Analyzer also determines setup and hold requirements at device pins, as well as maximum clock frequency. Critical paths identified by the Timing Analyzer can be highlighted in the editors.

Device Programming

All hardware and software necessary for programming, verifying and functionally testing EPLDs is available from Altera. The programming hardware includes an add-on card (for IBM PC-AT, PS/2, or compatibles) that drives the Master Programming Unit (MPU). The MPU supports functional testing, so that vectors developed during simulation can be applied to the EPLD at programming time to verify the functionality of the device. The MPU also performs continuity checking to ensure adequate electrical contact between the programming adapter and the EPLD.

In addition, Data I/O and a variety of PLD programmer manufacturers provide programming support for Altera EPLDs.

4.5. The Future

In the preceding sections we have traced the evolution of the EPLD beginning with the multi-array EP1800 and continuing through to the present day EPM7000 E-squared devices. One of the common architectural treads has been the multi-array style of architecture consisting of groups of logic elements (macrocells) which we called LABs, interconnected with a programmable routing structure which we called the PIA (programmable interconnect array). The use of EPROM technology allowed high-density, richly interconnected devices to be produced. The use of EEPROM technology allows for such devices to be programmed either off-line (as has traditionally been the case) or while in the end system. Thus, one can envision systems of EEPLDs which can be modified while in the system in the same way as is possible for SRAM programmable FPGAs.

At the same time, other programming elements, such as anti-fuses and SRAM bits, can be brought to bear on essentially LAB/PIA style architectures. One example of a new architecture currently emerging is the FLEX (Flexible Logic Element MatriX) product family from Altera. This family of devices contains many of the architectural characteristics of their predecessor EPLDs but are built on a standard CMOS technology. The programming elements for these devices are SRAM bits. A brief description of the FLEX architecture follows.

4.5.1 Functional Description

The FLEX architecture (Figure 4.5.1) incorporates a matrix of logic building blocks called logic elements (LEs). Each LE contains a four-input look-up-table (LUT) that provides combinatorial logic capability and a programmable register that offers sequential logic capability. Eight LEs are grouped together to form a Logic Array Block (LAB). LABs are arranged in rows and columns across the chip. I/O elements (IOEs) are located at the ends of rows and columns. Each IOE contains a bi-directional I/O buffer and a flip-flop that can be used as either an input or output register. Signal interconnections between LABs and to-and-from device pins are provided by an interconnect structure (called "FastTrack") made up of continuous metal lines that run the entire length and width of the device. In addition, the eight logic elements that make up a LAB communicate with each other through their own local interconnect structure without affecting the routability from one LAB to another.

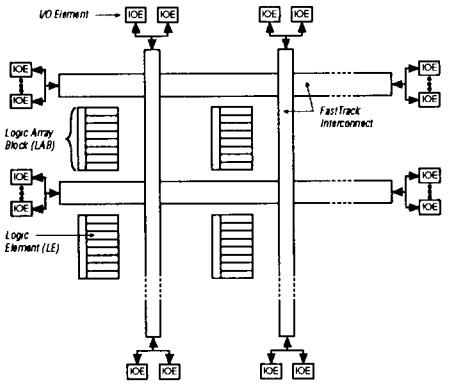


Figure 4.5.1 FLEX Device Block Diagram

Logic Elements

In addition to the LUT, each LE contains a programmable flip-flop, a carry chain, and a cascade chain. Figure 4.5.2 shows a block diagram of the LE. The programmable flip-flop in the LE can be configured for D, T, JK, or SR operation. The Clock, Clear, and Preset control signals on the flip-flop can be driven by dedicated input pins, general-purpose I/O pins, or any internal logic. For purely combinatorial functions, the flip-flop is bypassed, and the output of the LUT goes directly to the output of the LE.

Two fast data paths, the carry and cascade chains, connect adjacent LEs without using local interconnect paths. The carry chain provides a fast (less than 1 ns) carry-forward function between LEs. The carry from a lower-order bit moves forward into the higher-order bit via the carry chain, and feeds into both the LUT and the next portion of the carry chain. This feature allows the FLEX architecture to implement high-

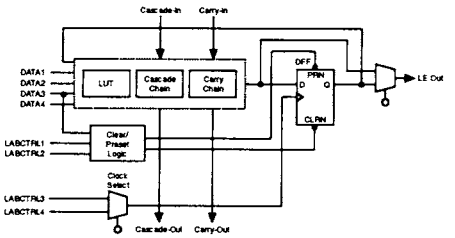


Figure 4.5.2 FLEX Logic Element (LE)

speed counters and adders of arbitrary width.

Figure 4.5.3 shows how an n-bit full adder can be implemented in n+1 LEs by using the carry chain. One portion of the LUT generates the sum of two bits using the input signals and the carry input; the sum is routed to the output of the LE. The register is typically bypassed for simple adders, but can be used for an accumulator function. Another portion of the LUT and the carry chain logic generate the carry, which is routed directly to the carry input of the next-higher-order bit. The final carry out is routed to an LE, where it can be used as a general-purpose signal.

The cascade chain provides for implementation of functions that have a very wide fan-in. The cascade chain uses a logical AND or logical OR to connect the outputs of adjacent LEs. Each additional LE provides four more inputs to the effective width of a function, with a delay of approximately 1 ns per LE. Figure 4.5.4 shows how the

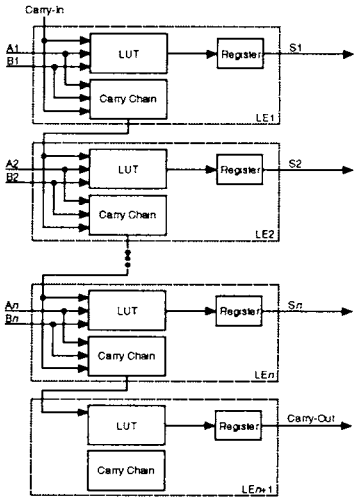


Figure 4.5.3 Carry Chain Operation

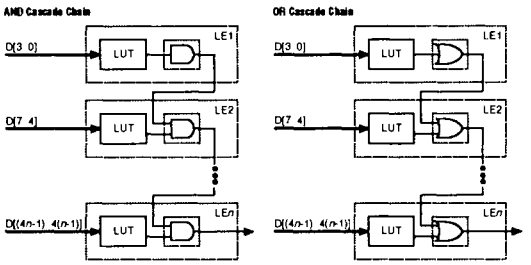


Figure 4.5.4 Cascade Chain Operation

cascade function can connect adjacent LEs to form functions with a wide fan-in. This example shows a function of 4n variables implemented with n LEs.

Logic Array Block

A Logic Array Block (LAB) consists of eight LEs, their associated carry and cascade chains, LAB control signals, and the LAB local interconnect. The LAB provides the coarse-grained structure of the FLEX architecture for efficient routing with high device utilization and high performance. Figure 4.5.5 shows a block diagram of the FLEX LAB.

Data signals enter the LAB local interconnect from either the row interconnect or the dedicated inputs. The outputs of all eight LEs are also driven back into the LAB local

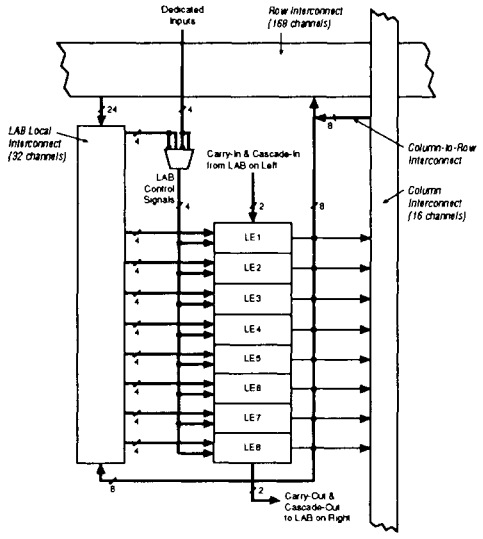


Figure 4.5.5 Logic Array Block (LAB)

interconnect via local feedback lines. Each LE in the LAB can drive signals out to the rest of the device via a row or column FastTrack interconnect path.

Each LAB provides four control signals that can be used by all eight LEs. Two of these signals can be used as Clocks, the other two as Clears. One Clear signal can also be used as a Preset. The LAB control signals can be driven directly from a dedicated input pin, an I/O pin, or any internal signal via the LAB local interconnect. The dedicated inputs are typically used for global Clock, Clear, or Preset signals because they provide synchronous control with very low skew across the device. If logic is required on a control signal, it can be generated in one or more LEs in any LAB and driven into the local interconnect of the target LAB. Programmable inversion is also available for all four LAB control signals.

FastTrack Interconnect

Connections between LEs and device I/O pins are provided by the FastTrack interconnect, a series of continuous horizontal and vertical routing paths that traverse the entire FLEX device. This device-wide routing structure provides predictable performance even in complex designs. In contrast, the typical routing structure of FPGAs consist of line segments stitched together through switch boxes, resulting in increased delays between logic resources.

Each row of LABs has a dedicated row interconnect that routes signals both into and out of the LABs in the row. The row interconnect can then drive I/O pins or feed other LABs in the device. Each column of LABs has a dedicated column interconnect that routes signals out of the LABs in the column. The column interconnect can then drive I/O pins or feed into the row interconnect to route the signals to other LABs in the device. A signal from the column interconnect, which can be either the output of an LE or an input from an I/O pin, must transfer to the row interconnect before it can enter an LAB. Figure 4.5.6 shows the interconnect of four adjacent LABs, with row, column, and local interconnects, as well as the associated cascade and carry chains.

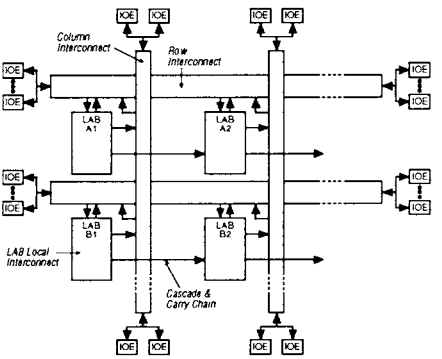


Figure 4.5.6 FLEX Interconnect Resources

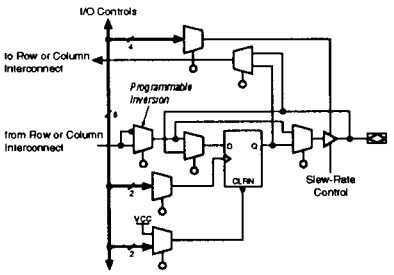


Figure 4.5.7 I/O Element (IOE)

I/O Element

Figure 4.5.7 shows the I/O element (IOE) block design. Signals enter the FLEX device from either the I/O pins that provide general-purpose input capability or the four dedicated inputs that are typically used for fast, global control signals. The IOEs are located at the ends of the row and column interconnect.

I/O pins can be used as input, output, or bidirectional pins. Each I/O pin has a register that can be used either as an input register for data that requires fast set-up times, or as an output register for data that requires fast clock-to-output performance. Each IOE has an adjustable output slew rate that can be configured for very low-noise or very high-speed performance. Designers can specify the slew rate on a pin-by-pin basis during design entry or assign a slew rate to all pins on a global basis.

The Clock, Clear, and Output Enable controls for the IOEs are provided by a network of six I/O control signals. These signals can be supplied by either the dedicated input pins or internal logic. The IOE control-signal sources are buffered to minimize skew across the device.

4.5.2 Configuration

The process of physically loading the SRAM programming data into the device is called configuration. Initialization occurs immediately after configuration. The initialization procedure resets registers, enables I/O pins, and causes the device to begin operating as a logic device. The configuration and initialization processes together are called "command mode;" normal device operation is called "user mode."

Configuration typically occurs immediately after the device is powered up but may also occur upon command. Real-time reconfiguration is performed by forcing the device into command mode with a device pin, loading different programming data, re-initializing the device, and resuming user-mode operation. The entire reconfiguration process requires less than 100 ms and can be used to dynamically reconfigure an entire system.

4.6. Design Applications

4.6.1 MAX 5000 Timing

Introduction

This section discusses internal delay paths, their relationships to AC specifications (shown in the MAX 5000 data sheets), and the calculated timing delays generated by MAX+PLUS II. Timing models for analyzing delays calculated by MAX+PLUS II, and equations that are used to calculate the delays are discussed.

EPLD Delay Parameters

Internal delays within an EPLD are described by a number of AC parameters (called microparameters) that refer to the actual internal delay within the device. Figure 4.6.1.1 shows the timing model for multiple-LAB devices. The following is a list of these microparameters and examples of how to predict timing delays with equations that use these microparameters as variables.

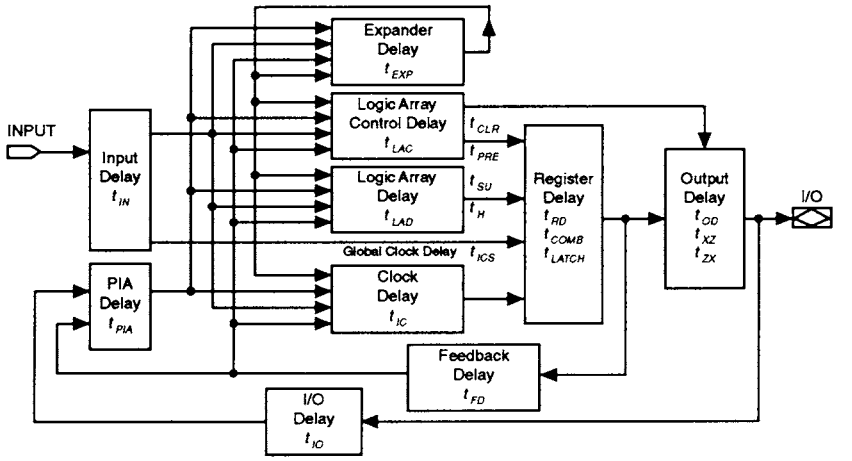


Figure 4.6.1.1 Timing Model for Multiple-LAB EPLDs

tIN Input pad and buffer delay. This delay directs the true and complement input signals from the dedicated input pin into the LAB. Within the LAB, the signals may propagate to any of four arrays: expander product-term array, logic array, logic array control, and clock array.

tIO I/O input pad and buffer delay for I/O pins used as inputs. When an I/O pin is used as an input, the tIO delay value must be added to tPIA to obtain the total delay from the I/O pin to the LAB.

tEXP Expander product-term array delay. This is the delay through the AND-NOT

structure of the expander product-term array. It is added to the delay already present in the four arrays when expanders are used, or added to itself when an expander feeds another expander.

tLAC Logic array control delay. This is the delay through the AND array by Clear, Preset, and Output Enable signals, representing the time required to propagate through the AND array to the CLRN and PRN inputs to the register, and the OE signal to the tri-state buffer.

tCLR Asynchronous register clear time, which represents the time required to reset a register output to a logical low. It is the time the register CLRN input is asserted low to the time the register output stabilizes at logical low.

tPRE Asynchronous register preset time. This delay represents the amount of time required to set a register output to a logical high. It is the time the register PRN input is asserted low to the time the register output stabilizes at logical high.

tLAD Logic array delay. This is the time a signal requires to propagate through a macrocell's AND array, the three-input OR gate, and the two-input XOR gate.

tICS System clock delay. This is the delay from the dedicated clock pin to a register's clock input.

tIC Clock delay. This is the delay through a macrocell's clock product term to the register clock input.

tFD Feedback delay. This delay is the propagation time from a macrocell output to any of the LAB's arrays, or the propagation time from a macrocell output to a PIA input or other macrocells in the LAB.

tSU Setup time required for a signal to be stable at the register input before the clock's rising edge.

tH Hold time required at the register input after the register clock's rising edge to ensure that the register stores the input data.

tRD Delay from the register clock's rising edge to the time that output appears at the register output.

tCOMB Combinational buffer delay, which is used only for combinational logic. This is the delay from the time the logic array's XOR output bypasses the programmable register to the time it becomes available for the macrocell output.

tLATCH Propagation delay through the latch from latch input to output.

tOD Output pad and buffer propagation delay from the macrocell output through the tri-state output buffer to the output pin.

tXZ Delay required for high impedance to appear at the output pin after the output buffer's active-high enable control is asserted low.

tZX Delay required for the macrocell output to appear at the output pin after the output buffer's active-high enable control is asserted high.

tPIA Programmable Interconnect Array delay for multiple-LAB devices. This delay is used for designs that use the PIA for routing. The PIA delay path starts where the macrocell feedback or I/O delay path ends, and ends where it enters the LAB and reaches any of its four arrays.

Critical pin-to-pin delay calculations are shown in Figures 4.6.1.2. The calculations used to derive these values from the microparameters listed in the MAX EPLD data sheets are shown for each path. These calculations assume that a dedicated input pin is used.

If the input comes from an I/O pin, tIO is substituted for the tIN value. For multiple-LAB EPLDs that use an I/O pin as an input, tIO + tPIA is substituted for the tIN value. If an expander is used in the path at any time, the tEXP value must also be added to the total delay path.

Example: 4-Bit Counter

Figure 4.6.1.3 depicts a synchronous 4-bit counter. The counter has one clock input (CLK) and the following outputs: RCO, QD, QC, QB, and QA. In addition, it has five inherent delays associated with registered logic (clock delay, input delay, array delay, feedback delay, and output delay) as well as setup time and hold time requirements for each register.

The propagation delay from CLK input pin to the clock input of the registers is tIN+ tIC (see Figure 4.6.1.3). If an I/O pin is used for input, the propagation delay from the I/O pin to the register's clock input is tIO + tIC, or tIO + tPIA + tIC for MAX EPLDs with multiple LABs. Since the delay from register to output pin is tRD + tOD, the total clock to output delay is tIN + tIC + tRD + tOD for dedicated input to output; tIO + tIC + tRD + tOD for I/O pin to output for MAX 5000-series EPLDs with one LAB; or tIO + tPIA + tRD + tOD for I/O pin to output for MAX 5000-series EPLDs with multiple LABs.

In addition, data input to the register must meet both setup and hold time requirements. The internal setup time is the time needed for the input data to stabilize before the triggering edge of the clock appears at the register input. The external setup time is the algebraic difference between the sum of the input, logic array, and setup time, and the sum of the input and clock delay: (tIN + tLAD) - (tIN + tIC) + tSU. When expanders are used, tEXP must be added, and when I/O pins are used, tIO or (tIO + tPIA) must be added. As long as the external setup time is met at the inputs, the counter functions properly.

The maximum internal counter frequency (fCNT) is the inverse of tCNT , which is the worst-case delay for internal feedback. This frequency is the minimum internal clock period at which the counter can operate correctly. tCNT is the sum of delay

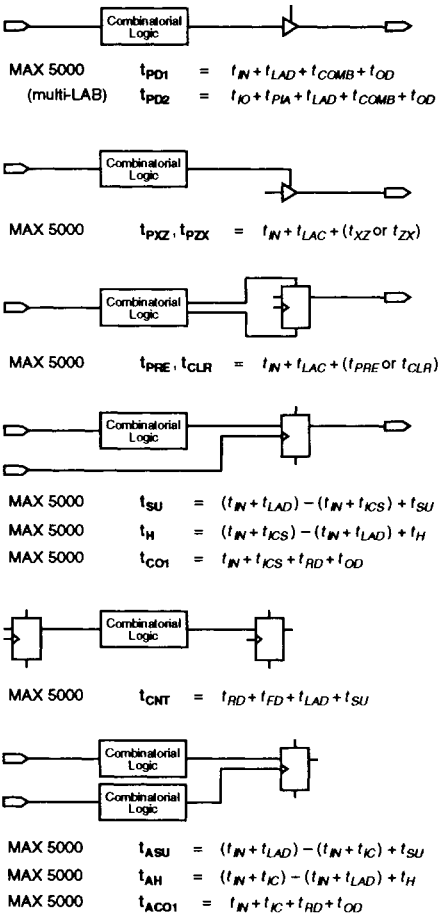


Figure 4.6.1.2 Critical Pin-to-Pin Delay Calculations

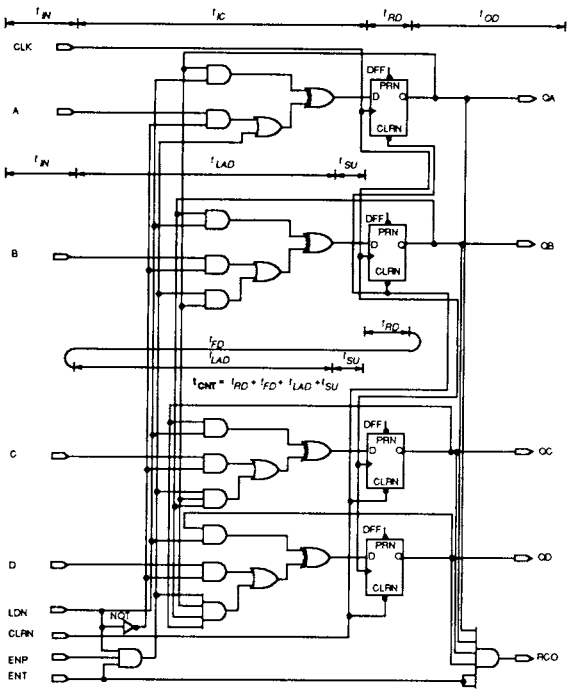


Figure 4.6.1.3 Timing Analysis of 74161 Counter

paths that the register feedback must traverse before reaching a register input and meeting the internal setup time: (tRD + tFD + tLAD + tSU). Once the clock triggers QB, data takes tRD delay prior to appearing at the register output. The signal feeds back (tFD) and flows through the logic array (tLAD). Finally, the signal reaches the register QC and meets the setup time of the register (tSU).

When expanders are used, tEXP must be added as the signal passes through the expander array before reaching the logic array. The tCNT delay represents only internal circuit delays, while a circuit that depends on external and internal signals must also account for input and I/O delays. For example, the internal path (tCNT) is 20 nsec for EPM5128-1, while the external path (tSU + tCO) is 29 nsec.

4.6.2 Using Expanders to Build Registered Logic in MAX EPLDs

Introduction

Each EPLD macrocell contains one register that can be programmed for registered functions or bypassed for combinational functions. However, some applications require more registers than are available in the macrocells. These extra registers can be built with expander product terms (expanders). This section explains how and

when to use expander latches and registers, and describes timing considerations, specifically for an SR latch, a transparent D latch, and a synchronous register.

Expander product terms

Expanders are unallocated product terms with inverted outputs that feed the logic array. Each expander is fed by the same inputs that feed the macrocell: a global bus, macrocell feedbacks, other expanders, and I/O feedbacks. Since expanders feed themselves, they can be used to build latches and registers. Two expanders (EXP primitives) can be cross-coupled to generate an SR latch, three can be used to build a transparent D latch, and six can be used to build a synchronous D flip-flop with asynchronous Preset and Clear. The expander circuits described here have been built into macrofunctions to optimize performance. Each function is built with AND gates and EXP primitives to optimize fitting.

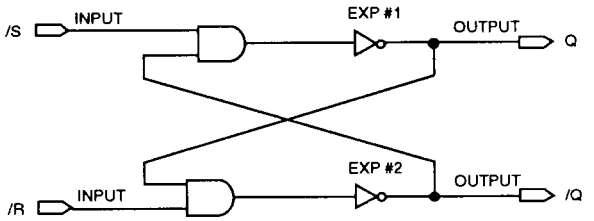


Figure 4.6.2.1 SR Latch Implemented with Expanders

Asynchronous SR Latch

Figure 4.6.2.1 shows an asynchronous SR latch implemented with two expanders. Since expanders are product terms with inverted outputs, the latch is a NAND implementation with the Set and Reset terms active low. If both inputs are simultaneously low, both outputs will become logic high until either of the inputs goes high.

The functional output of an SR latch is shown in Table 1. To implement the latch with active-high inputs, as in a NOR latch, the inputs are inverted with NOT primitives. Asynchronous SR latches are often used to debounce input-switching circuits or to detect edges in switching circuits.

Table 1

/S	/R	Q
H	H	Q0
L	H	H
H	L	L
L	L	H (1)

Note:(1) /S and /R low causes both Q and /Q to be high.

SR Latch Timing

Each expander has a timing delay defined as tEXP. The hold time for the SR latch shown in Figure 4.6.2.2 is given as tH = 2* tEXP. A low signal at the S input must remain low long enough to propagate through Expander 1 and Expander 2 to latch the input. The propagation delay from the latch input to the latch output is given as tCO = 2* tEXP. A low at the S input must travel through Expander 1 and Expander 2 before the outputs Q and /Q become valid.

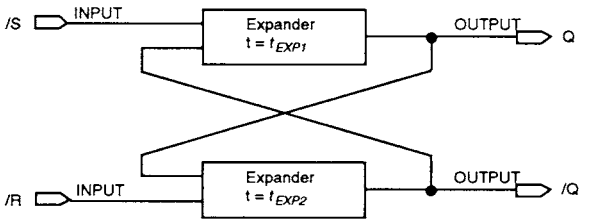


Figure 4.6.2.2 SR Latch Timing Model

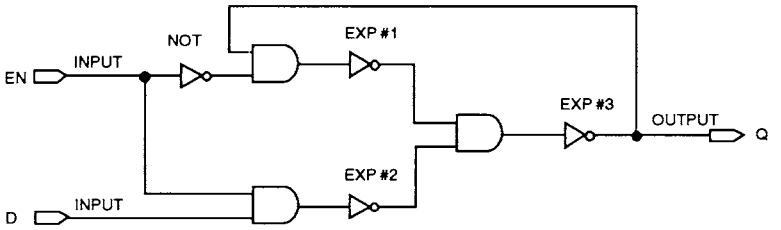


Figure 4.6.2.3 Transparent D Latch Implemented with Expanders

Transparent D Latch

The transparent asynchronous D latch with EN (Enable) is implemented with three expanders, as shown in Figure 4.6.2.3. This latch is functionally comparable to a 74LS373. The latch is transparent when EN signal is logic high. When EN goes low, the input is latched. This latch is especially applicable for latching inputs from a bus. The functional output of this latch is shown in Table 2.

Table 2

EN	D	Q
L	L	Q0
L	H	Q0
H	L	L
H	H	H

Transparent D Latch Timing

The delay paths for the transparent asynchronous D latch are shown in Figure 4.6.2.4. The tH value for this circuit is 0 ns because the paths from the D input and the EN input have delays equal to those of Expander 3, which latches the result. Both the set up time tSU and the enable to output time tCO is given as two expander delays. The setup time, tSU, requires the D input to go through Expander 2 and Expander 3 to reach Expander 1 before it can be latched. tCO is the delay through Expander 2 and Expander 3 to the output from the rising edge of EN.

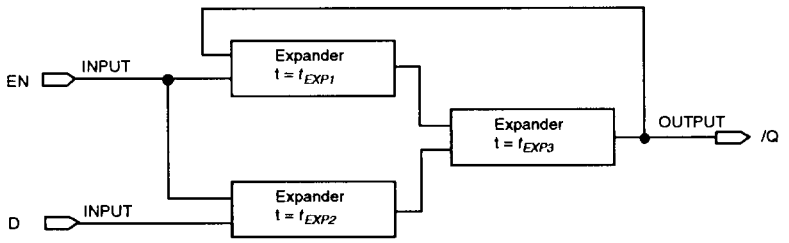


Figure 4.6.2.4 Transparent D Latch Timing Model

Synchronous D Register

The function outputs for the synchronous D register are shown in Table 3. A synchronous D register with asynchronous Preset and Clear (/P and /C) can be built with six expanders, as shown in Figure 4.6.2.5.

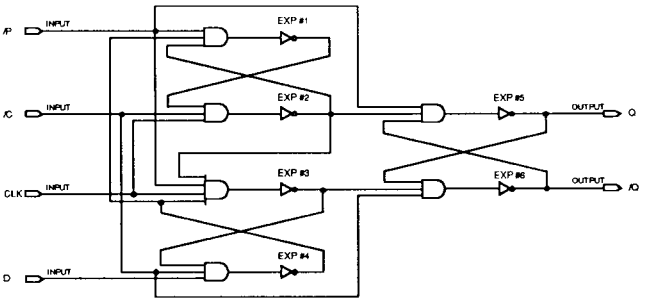


Figure 4.6.2.5 Synchronous D Register with Preset and Clear

Table 3

/P	/C	D	CLK	Q	/Q
H	H	L	U	L	H
H	H	H	U	H	L
H	H	X	H	Q0	/Q0
H	H	X	L	Q0	/Q0
L	H	X	X	H	L
H	L	X	X	L	H
L	L	X	X	H	H

The state at the D input is clocked into the latch with a rising edge at the clock input. Both the true and complement signals are available at the output. This output remains latched until the next rising edge clock or until the Preset or Clear is asserted low. The Preset and Clear can be made active high by placing NOT primitives in front of the two signals. Using expanders as registers increases the total register count in a given MAX 5000 device by 31%. For example, the EPM5192 can have up to 60 registers implemented with expanders, which gives it a total capacity of 252 registers. However, the speed of these expander registers will be slower than the macrocell registers.

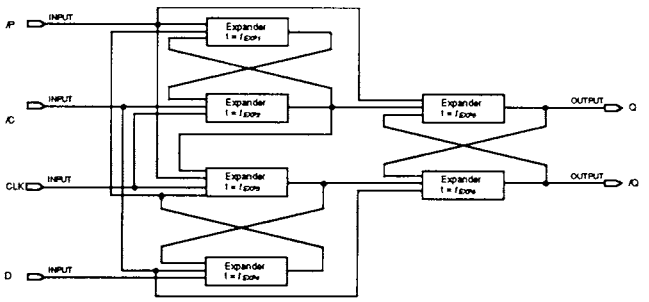


Figure 4.6.2.6 Timing Model for Synchronous D Register

Synchronous D Register Timing

The timing paths for the synchronous D register are shown in Figure 4.6.2.6. Two different timing paths exist for the D input, depending on whether the D input is high or low. The worst-case path is described here. The worst case th is one expander delay through Expander 3. The worst case for tsu is the path through Expander 4 and Expander 1 to the input of Expander 2. The worst case clock-to-output (called tco1) path is through Expander 3, Expander 6, and Expander 5 for both Q and /Q to produce valid outputs. Both tclr and tpre have a delay path through Expander 5 and Expander 6 to produce a valid output. tcnt is the minimum recirculation time for the register which is tco1 + tsu or five expander delays. The inverse of tcnt is fcnt or the

maximum toggle frequency.

Fitting Expanders Into MAX Designs

Latches and registers built using expanders can provide valuable additional resources. Their use, however, must be carefully analyzed to ensure correct setup time, hold time and tco. These expander latches and registers are particularly useful for latching input signals. They should not be used when the D input is a complex function because other expanders are the only source of this logic. Instead, registers driven by complex logic should be placed in macrocells. On the other hand, if additional logic is required after the register, expander registers should be used, since the output feeds directly into the logic within the macrocells.

4.6.3 Simulating Internal Buses in General-Purpose EPLDs

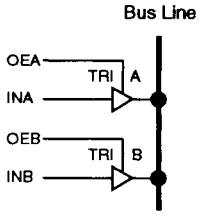
Introduction

Altera's EPLDs allow internal buses to be emulated by using logic to replace tri-state functions. A series of simple multiplexers can create buses with several sets of input signals. Multiplexing also saves device resources and helps to eliminate timing and loading problems. This section describes how to use multiplexers for different bus configurations and explains the benefits of this approach.

Using multiplexing to emulate tri-state functions saves macrocells and I/O pins for applications that would otherwise require a bus external to the EPLD. Figure 4.6.3.7 shows a 4-to-1 multiplexer in a single macrocell that emulates a bus line with four sources. With conventional tri-stating techniques, the same function requires four macrocells and I/O pins, as shown in Figure 4.6.3.8. Multiplexing saves three macrocells and I/O pins if the switching functions are implemented with the product terms inside the macrocell, instead of with tri-state buffers and I/O pins external to the macrocell.

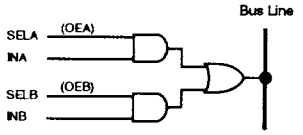
Two-Source Bus Configurations

Figure 4.6.3.1 shows the simplest bus configuration, a one-bit bus created by connecting the outputs of two tri-state buffers to a single line. The function table shows the possible states of the bus. When tri-state buffer A is enabled, the input to



INA	OEA	INB	OEB	BUS LINE
X	0	X	0	Z or 1
X	0	0	1	0
X	0	1	1	1
0	1	X	0	0
1	1	X	0	1

Figure 4.6.3.1 One-Bit Bus



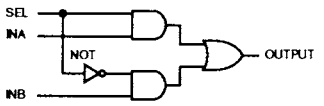
INA	SELA	INB	SELB	BUS LINE
X	0	X	0	0
X	0	0	1	0
X	0	1	1	1
0	1	X	0	0
1	1	X	0	1

Figure 4.6.3.2 AND/OR Logic Emulating Tri-State Functions

that buffer (INA) appears on the bus. When tri-state buffer B is enabled, the input to that buffer (INB) appears on the bus. If neither buffer is enabled, the bus is in a high-impedance, or floating, state. Such buses are often tied high with a pull-up resistor to prevent them from floating.

Figure 4.6.3.2 shows two AND gates and an OR gate that emulate the tri-state functions of Figure 1. Each AND gate has a data input (INA or INB), and a select input (SELA or SELB) that represents the original Output Enable control. The function table shows that the AND/OR logic exactly emulates the original tri-state functions, if one of the two outputs is always selected. If neither output is selected, the output of the AND/OR logic is low.

The select controls are mutually exclusive, since only one input is ever enabled onto a bus at any given time. Therefore, they can be encoded into a single input by making SELA the common select input, and then feeding the inverse of this signal into the previous SELB input as shown in Figure 4.6.3.3.



INA	INB	SEL	OUTPUT
X	0	0	0
X	1	0	1
0	X	1	0
1	X	1	1

Figure 4.6.3.3 Multiplexer Created with AND/OR Logic and Select Controls

Additional 2-to-1 multiplexers, all controlled by a common select signal, can create wider buses. One multiplexer is necessary for each bit of the bus. For example, Figure 4.6.3.4 shows eight 2-to-1 multiplexers emulating a byte-wide bus.

Buses with Three or More Sources

Larger multiplexers with multiple select inputs can emulate buses with more than two sources. Figure 4.6.3.5 shows how a 4-to-1 multiplexer can create a bus with up to four sources. Figure 4.6.3.5 also includes a truth table with the proper encoding for the select inputs. This type of multiplexer can also implement buses with two or three sources.

Additional multiplexers with shared select lines can create buses with nearly any width. For example, five 4-to-1 multiplexers can create a 5-bit-wide bus with two, three, or four sets of inputs.

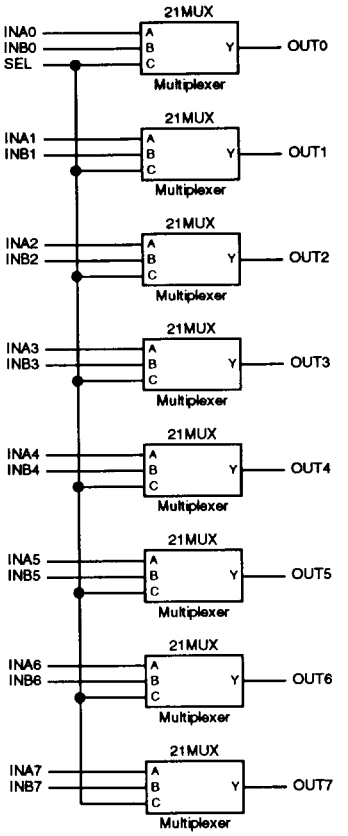


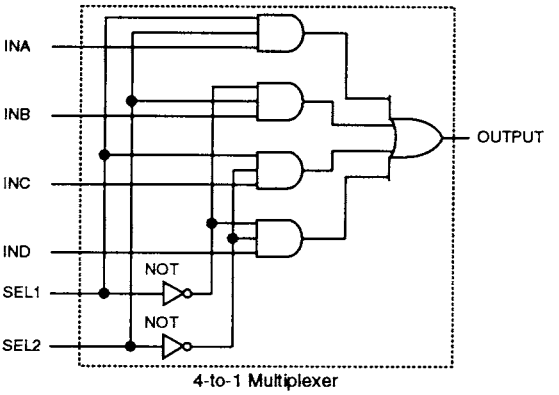
Figure 4.6.3.4 Eight 2-to-1 Multiplexers Emulating a Byte-Wide Bus

Implementing Bus Functions with Hardware Description Language

Hardware Description Languages, such as Verilog, VHDL (VHSIC Hardware Description Language), or AHDL [Altera AN22 1990] provide a quick alternative to graphic schematic entry for implementing bus functions with multiplexing. HDLs can be used to describe buses with nearly any number of inputs and of nearly any width.

For example, Figure 4.6.3.6 (next page) shows the AHDL code required to create an eight-bit bus with three sources. The data inputs are A7 to A0, B7 to B0, and C7 to C0. The two select inputs, SEL1 and SEL2, can be treated as an encoded group in AHDL. These select lines control which set of input signals is connected to the outputs through a series of simple IF-THEN statements.

By adding data inputs (e.g., D[7..0]), this file can be easily modified to create a bus with more sources. For multiplexers with more than four data inputs, one more bit



INA	INB	INC	IND	SEL2	SEL1	OUTPUT
X	X	X	0	0	0	0
X	X	X	1	0	0	1
X	X	0	X	0	1	0
X	X	1	X	0	1	1
X	0	X	X	1	0	0
X	1	X	X	1	0	1
0	X	X	X	1	1	0
1	X	X	X	1	1	1

Figure 4.6.3.5 Four-to-One Multiplexer Implementing a Bus with up to Four Sources

```
SUBDESIGN BUSMUX (
   A[7..0],
   B[7..0],
   C[7..0],
   SEL[1..0]: INPUT;
   OUT[7..0]: OUTPUT; )
BEGIN
   IF (SEL[]==0) THEN OUT[]=A[]; END IF;
   IF (SEL[]==1) THEN OUT[]=B[]; END IF;
   IF (SEL[]==2) THEN OUT[]=C[]; END IF;
END;
```

Figure 4.6.3.6 AHDL Implementation of Eight-Bit Bus with Three Sources

must also be added to the SEL group (e.g., SEL[2..0]) for each factor-of-two increase in the number of data inputs. For example, a seven-input multiplexer requires three select bits.

The width of the bus can be varied by changing the input and output group widths. For example, the declaration A[5..0] creates a 5-bit wide set of A inputs.

PROs and CONs

Emulating tri-stated buses with logic eliminates timing hazards such as bus contention, which occurs when two or more tri-state outputs are simultaneously enabled onto a single bus line. This condition (usually unintended) can cause an unpredictable logic level to propagate if multiple buffers are driving high and low at the same time. The select controls for simple AND/OR logic (shown in Figure 4.6.3.2) can both be enabled at the same time, but the result will be a known logic level. The select controls can never be enabled at the same time if they are encoded, as in the true multiplexer configurations (Figure 4.6.3.3).

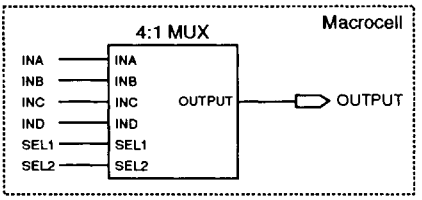


Figure 4.6.3.7 Four-to-One Multiplexer in a Single Macrocell

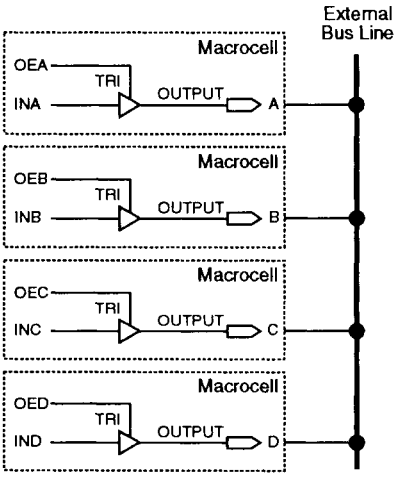


Figure 4.6.3.8 Four-to-One Multiplexer Implemented with Traditional Tri-State Logic

A potential timing hazard for a multiplexer configuration is output glitching caused by input signal skew. EPLD architecture minimizes skew difficulties and glitching is seldom a problem in actual designs. However, the designer must exercise care when driving edge-sensitive logic from multiplexer outputs. A potential disadvantage of replacing tri-state buses with multiplexers is that all signals must be routed to the single multiplexer location. If the function can be implemented in a single EPLD, this is usually not a problem. If the tri-state function is spread across several chips, then signal routing is a more serious problem.

Replacing tri-stated buses with logic reduces capacitive loading limitations. High fan-outs to traditional buses create high capacitive loads that reduce bus bandwidth. Macrocells and feedback paths in EPLDs have constant delays, regardless of the number of signals entering the macrocell. If control logic is implemented with multiplexers, internal loading is not a problem.

4.6.4 Fast Bus Controllers with the EPM5016

Today's advanced microprocessors are capable of running in systems with clock speeds greater than 33 MHz. To realize the performance potential of these microprocessors, their memory interfaces must be equally fast. High-performance memory devices, however, are expensive. Cost can be decreased with little impact on performance by using a combination of fast and slow memories. To accommodate the slower memories, wait states are added into the microprocessor bus cycle. A bus controller which contains the wait-state and bus-control logic for a 80386 system can be integrated into an Altera EPM5016 MAX EPLD. The EPM5016 has a propagation delay of 15 ns and can support a system clock rate of 66 MHz. With a specified output drive of 24 mA, it can be directly connected to buses.

Figure 4.6.4.1 shows a block diagram of an 80386 microsystem that incorporates peripheral logic, memory, and an 8259A interrupt controller. The EPM5016, shown in the center of the diagram, serves as the system bus controller. The EPM5016 decodes the 80386 status signals to control the peripheral logic, the data transceiver, interrupt controller, and other external logic. It also extends bus cycles by adding wait states to interface to slower peripherals and memory devices.

The 80386 halts processing to allow wait states to be added into the bus cycle when the signal /READY is high. (Note that the slash (/) is used to indicate an active-low signal.) The EPM5016 bus controller tracks each bus cycle operation and causes /READY to go high when wait states are needed. For example, read operations from 200-ns EPROM memory in 33-MHz systems require 14 wait states.

The EPM5016 also decodes the bus control signals IORD (I/O read), IOWR (I/O write), and INTA (interrupt acknowledge). The 24-mA output drivers on the EPM5016 eliminate the need to buffer these bus signals externally.

The 16 data signals originating from the 80386 are isolated from the system data bus

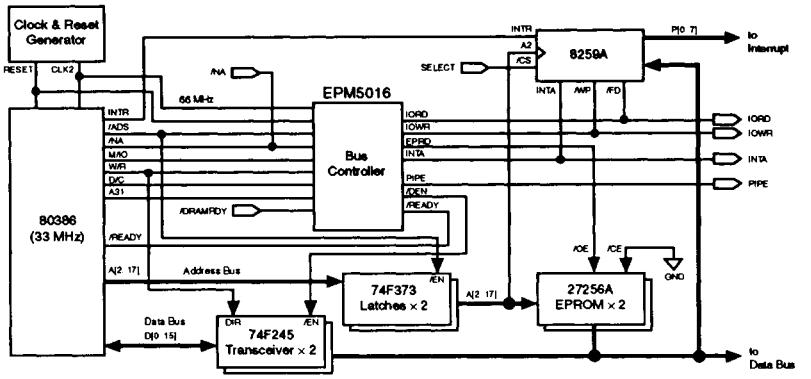


Figure 4.6.4.1 80386 Subsystem Block Diagram

with two 74245 8-bit transceivers. The tri-state control on the transceivers is provided by the signal /DEN from the EPM5016. The direction signal is controlled directly by the Read-Write signal (W/R).

Two 74373s (8-bit latches) are used to latch the 80386 address signals at the beginning of the bus cycle to maintain a valid address throughout the cycle. The latches are controlled by the 80386 signal /ADS. The high performance of the EPM5016 easily supports the 33-MHz bus cycles. In fact, a 66-MHz (2 x 33-MHz) clock is used to clock the design for two reasons. First, /ADS can be connected directly to the address latches since the EPM5016 control signals for the peripheral logic are active before the end of the first bus cycle. Second, the wait-state generator offers finer granularity with 15-ns cycles than with 30-ns cycles.

The design for the bus controller circuit requires 9 inputs: all 8 dedicated inputs of the EPM5016 and 1 I/O pin. Five of the 9 inputs to the EPM5016 are signals from the 80386 microprocessor. The functions of these 5 signals are given in Table 1. The other inputs are described subsequently.

Table 1. EPM5016 Input Functions

Input	Function
M/IO	Memory or I/O
W/R	Read or Write status
D/C	Data or Control status
A31	Address bit 31 for memory mapping of the EPROM
/ADS	Address data strobe indicating the beginning of the bus cycle

Systems that have functions set up for pipelining require external logic to generate the

signal /NA (next address). /NA feeds both the 80386 and the EPM5016 bus controller. When /NA is activated, the 80386 places the next address on the address bus so that it may be latched. Applications that require wait states receive minimal benefit from pipelining. In such cases, the /NA signal is used simply to disable the EPM5016.

/DRAMRDY is an externally generated signal that, when high, halts the 80386 by causing the /READY signal to be high. When /DRAMRDY goes low, the 80386 continues processing.

The 66-MHz clock (CLK2) feeds a toggle flip-flop in the EPM5016 to generate a divide-by-two signal (CLK) that matches the 33-MHz system clock signal. CLK tracks the microprocessor clock phase.

The last input signal, RESET, is connected directly to the reset signal of the microprocessor. RESET feeds the preset or reset of each register to set the EPM5016 to the correct start-up state.

Figure 4.6.4.2 shows the flow diagram for the bus controller. First, the 80386 causes /ADS to go low, indicating that a bus cycle has begun. /ADS then causes BUS_TRCK to activate the signal BUS_ACTIVE, which indicates that the processor is in an active bus cycle. BUS_ACTIVE feeds the functions DECODE and TRAN_CTL.TRAN_CTL then scans the 80386 control signals and enables the data transceiver buffers when they are required. DECODE also scans the 80386 control signals and decodes them into specific control signals (IORD, IOWR, EPRD, and INTA) that drive the peripheral logic and the block function, WAIT_CNT.WAIT_CNT is a loadable 4-bit counter that counts the required number of wait states for bus cycles. When WAIT_CNT finishes the wait-state count, it causes the TIME_DELAY to go low, enabling DECODE to release the /READY signal and finish the bus cycle.

4.6.5 Micro Channel Bus Master and SDP Logic with the EPM5032 EPLD

This section describes a Micro Channel bus master interface with streaming data transfer capability. A data transfer rate of 80 Mbytes/sec can be achieved by using full 32-bit address and data buses in the interface design. The essential bus master and streaming data functions are implemented in an Altera EPM5032 MAX EPLD using thirteen inputs and seven outputs. The logic is implemented as a synchronous state machine timed by a 20-MHz clock. The EPM5032 works together with the Altera EPB2001 and EPB2002A EPLDs to provide a complete Micro Channel bus master interface. The features of this implementation are listed below.

- Bus control logic for basic and streaming data transfer
- Support for basic transfer cycles
- -Default (200-ns cycle)
- -Synchronous extended (300-ns cycle)
- -Asynchronous extended (>300-ns cycle)

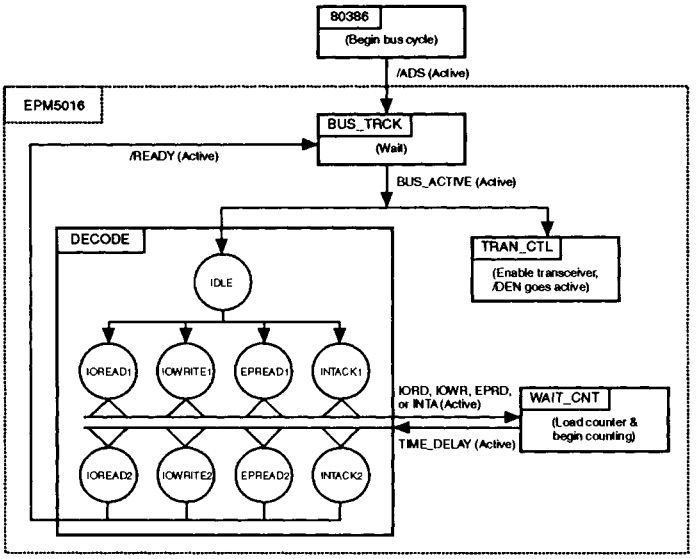


Figure 4.6.4.2 Bus Controller Flow Diagram

- Support for streaming data enable function (external POS bit)
- Support for streaming data cycles; (16-, 32-, and 64-bit)
- -Deferred streaming cycles
- -Data pacing
- -All modes of streaming data cycle termination
- Clock signal generation to track -SD STROBE during data pacing
- Bus master and SDP logic implementation as a synchronous state machine timed by a 20-MHz clock
- User-defined pin assignments allowed (except VCC and GND) and input polarities
- Available in 28-pin DIP, SOIC, or JLCC packages

Figure 4.6.5.1 illustrates the Micro Channel interface implemented with the EPB2001, EPB2002A, and EPM5032 EPLDs. The application-specific address control, address generation, and data steering functions are left to the designer for greater flexibility. The EPB2001 provides the I/O or memory-slave interface, and the EPB2002A provides DMA arbitration functions. The EPM5032 implements bus master control logic with SDP capability.

When the EPB2002A signals to the EPM5032 that the bus has been won, the EPM5032 generates the essential bus control signals. If streaming data capability is enabled (via any user-configurable POS register bit from an EPB2001 POS I/O pin)

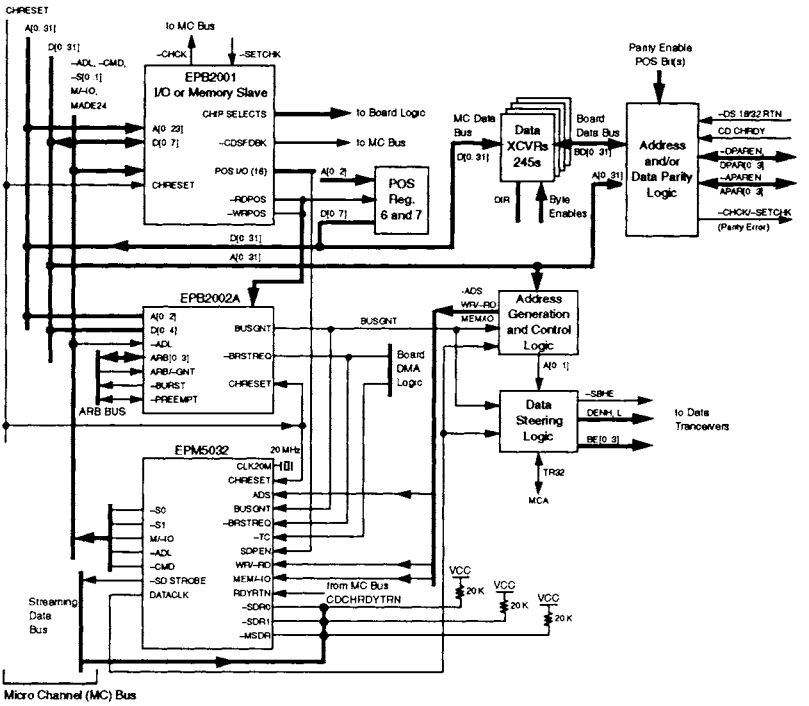


Figure 4.6.5.1 Micro Channel Bus Master Interface

the EPM5032 detects whether the slave can perform streaming operations and, if so, generates the -SD STROBE signal and DATA_CLK output. The DATA_CLK output tracks the -SD STROBE signal. It uses the RDYRTN signal to provide a "stretched out" clock for internal use on the board (e.g., for use by an address increment counter).

Bus control for burst transfers is indicated to the EPM5032 by the EPB2002A signals BUSGNT and -BRSTREQ. BUSGNT is an EPB2002A output which signals that the bus has been won. An active -BRSTREQ input to the EPB2002A indicates that the bus was requested by the adapter for burst transfers.

The -ADS input is required to determine when addresses are valid on the address bus. It is similar to the -ADS signal on the 80386 processor, which indicates the start of the cycle, and is used by the EPM5032 to latch the state of the WR/-RD and MEM/IO signals. After these signals are latched, the EPM5032 drives the -SO, -S1, M/-IO, -ADL, and -CMD signals. The -SO and -S1 status signals are decoded from the state of the WR/-RD signal; M/-IO is decoded from the state of the MEM/IO input.

The bus master and streaming data logic are implemented in AHDL as a synchronous state machine. Figure 4.6.5.2 shows a diagram of this state machine. It is timed by a

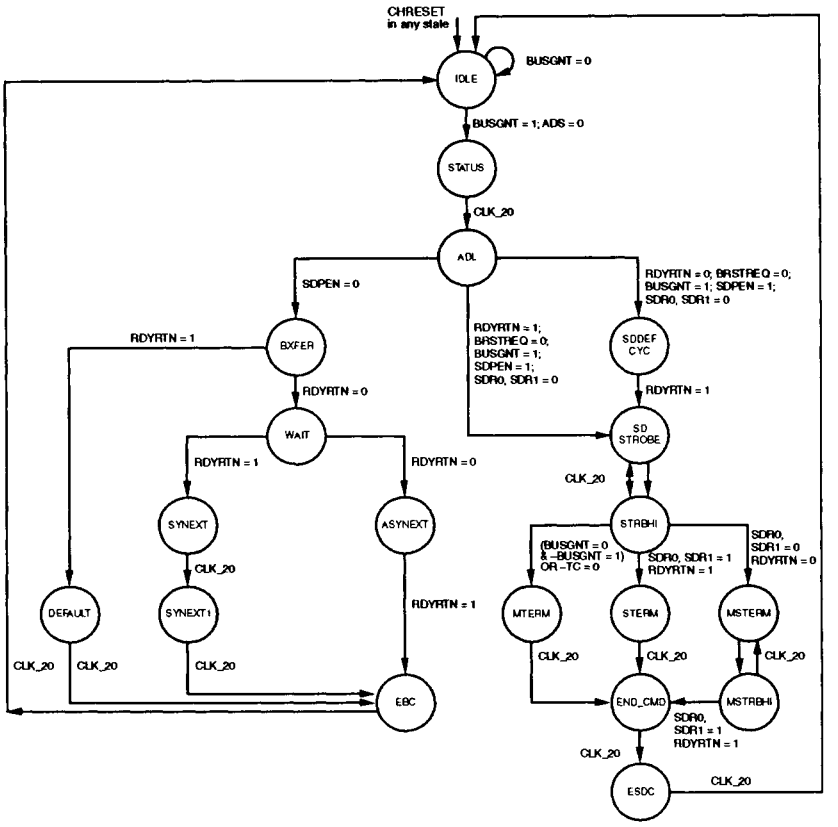


Figure 4.6.5.2 Bus Master SDP Controller State Machine

20-MHz clock fed on the CLK_20 input pin. Outputs of the state machine are decoded as functions of the present state. The next state is determined by the current state and inputs.

4.6.6 FIFO Controller Using an EPM7096

Interacting digital subsystems often consume and produce data at different rates and times. A First In/First Out (FIFO) buffer in a data path can bridge this rate and time mismatch so systems can function correctly. With the appropriate FIFO buffer, data written into the FIFO can be read in the order it was entered, but at an independent rate. This section describes how to use the 96-macrocell EPM7096 to design an 8 K x 16 bit FIFO.

Two types of FIFO buffers are available: shift FIFOs and pointer FIFOs. Shift FIFOs

are typically constructed from a linear array of registers, where the number of registers is the same as the number of bits in the data word. Data to be stored in the FIFO is written to the first location in the register array. This data then "bubbles" to the last empty location nearest the output. When data has shifted to the last location of the register array, it can be read. The major drawback of shift FIFOs is fall-through time delay, which is the time required for valid data to propagate from the input to the output of an empty FIFO.

In contrast, pointer FIFOs do not have this fall-through time delay. Data is stored in a Random Access Memory (RAM) array. Read pointers store the next address to be read from this array; write pointers store the next address to be written to this array. If these pointers are made up of counters that roll over (i.e., restart at 0) at a value equal to the size of the FIFO, the FIFO structure becomes a circle. See Figure 4.6.6.1.

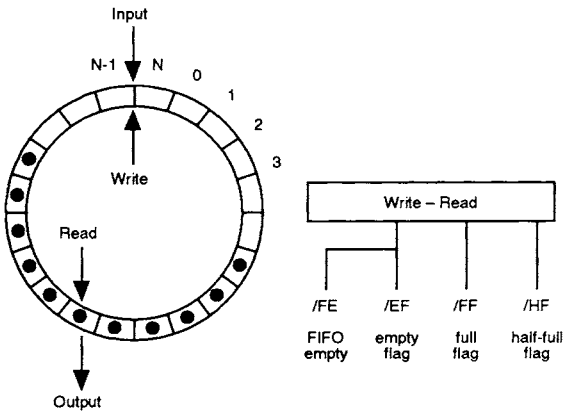


Figure 4.6.6.1 Circular Pointer FIFO

To enable the designer to determine whether the FIFO is full, empty, or at any point in between, the following Up/Down counter can be added to the design:

amount of data = (number of writes) - (number of reads)

Pointer FIFOs are available in a single package that contains read and write pointers and a RAM array. With this option, however, the size of the RAM arrays and the type of control functions available are limited. A designer can get around this limitation by using a standard static RAM device and implementing the control logic in an EPLD to maximize buffer-size options and meet custom interface-control requirements. The EPLD/RAM solution also significantly reduces cost. An 8 K x 16 bit FIFO implemented with RAMs and an EPLD costs less than half the price of a comparable off-the-shelf solution with two 8 K x 8 bit FIFOs.

Figure 4.6.6.2 shows a typical application of a pointer FIFO buffer implemented with

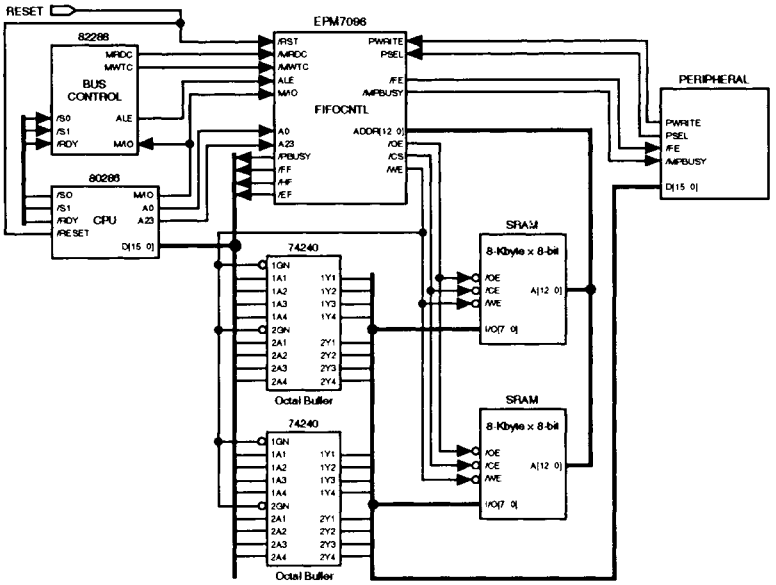


Figure 4.6.6.2 EPM7096 FIFO Application

an EPM7096 and two 8 K x 8 SRAMs. The input to the FIFO is a 10-MHz 80286 microprocessor. The FIFO controller interfaces to the /MRDC, /MWTC, M/IO, and ALE signals from the 80286 and 82288 bus controllers. Chip select is generated by decoding the address lines A0 and A23.

The microprocessor can read the FIFO status from the four active-low tri-stated lines that are connected directly to the microprocessor's data bus: /EF (empty flag), /HF (half full), /FF (full flag), and /PBUSY (peripheral busy). The microprocessor bus and the peripheral bus are isolated by a pair of 74240 octal buffers controlled by the /WE output of the EPM7096 FIFO controller.

In this example, an arbitrary peripheral is connected to the output of the FIFO. This peripheral selects the FIFO with the PSEL (peripheral select) line and strobes data out of the FIFO with the PWRITE (peripheral write) line. The peripheral determines the status of the FIFO by reading /FE (FIFO empty) and /MPBUSY (microprocessor busy) signals.

The two 8 K x 8 bit static RAM devices store up to 8192 16-bit data words (the same number of bits as the 80286 microprocessor's data bus). The EPM7096 FIFO controller provides the RAM with all required control and address lines, including /OE (output enable), /CS (chip select), /WE (write enable), and address lines A0 to A12.

Figure 4.6.6.3 shows the block diagram for the EPM7096 FIFO controller. Two 13-bit

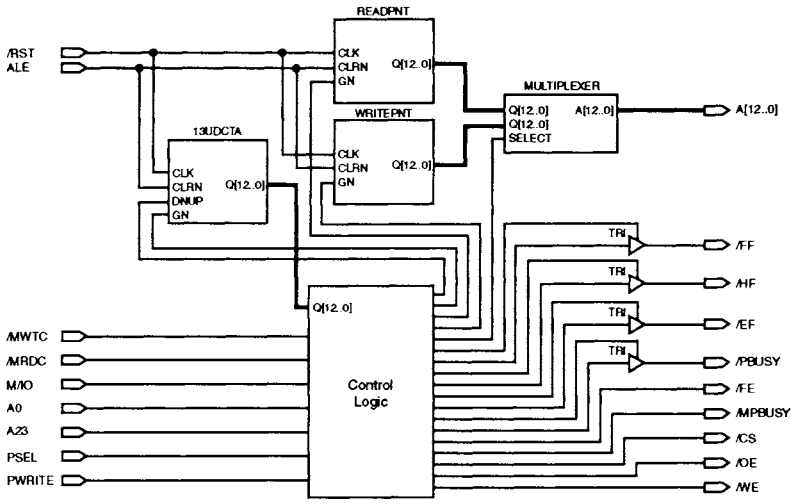


Figure 4.6.6.3 EPM7096 FIFO Controller

counters provide the read and write pointers for the EPM7096. The read pointer counter contains the address where the next read data is stored; the write pointer counter contains the address where the next write data is stored. A third 13-bit counter, 13UDCTA, determines FIFO status (13 bits define 8192 unique addresses).

When a FIFO read or write operation is required, a 26- to 13-bit address-pointer multiplexer selects the read or write pointer outputs and presents them to the RAM address bus. The control logic block decodes FIFO requests, provides FIFO status outputs, and controls the address-pointer multiplexer.

4.6.7 Integrating an Intelligent I/O Subsystem with a Single EPM5130 EPLD

When higher system performance is necessary, many designers first consider a faster microprocessor or a new microprocessor architecture. In many cases, however, sufficient speed can be achieved through an intelligent I/O subsystem that has been optimized for a particular task. Transferring I/O processing to intelligent subsystems also allows the system processor to dedicate more processing power to primary system functions.

This section describes how the custom logic requirements of such a subsystem can be integrated into a single Altera EPM5130 MAX EPLD, replacing over 75 standard TTL packages.

Figure 4.6.7.1 shows a subsystem using the T1 serial coprocessor implemented in an EPM5130 EPLD. The serial coprocessor consists of two T1 serial transmitters, control and address generation logic for a buffer RAM, and the I/O subsystem control logic. The system processor writes data for serial transmission through the T1 serial coprocessor into the buffer RAM. When data transfer is complete, the system processor signals to the serial coprocessor that data can be transmitted. The serial coprocessor then transfers the data to the transmitters for serialization. The buffer RAM control logic also features error detection and correction. Data can be sent over either T1 serial channel with individual variations in protocol. Once all of the data in buffer RAM is transferred, the processor is interrupted, and the cycle can repeat.

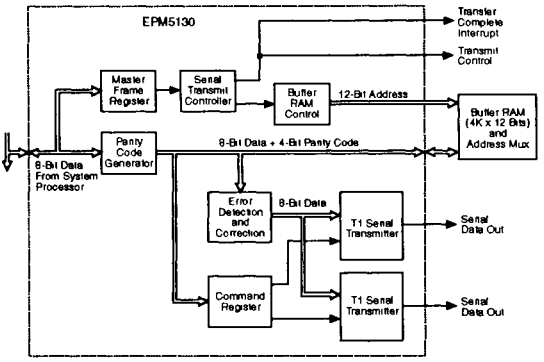


Figure 4.6.7.1 T1 Serial Coprocessor Block Diagram

The T1 serial coprocessor is a useful example of an intelligent I/O subsystem design because it contains the types of logic, such as decode and control state machines, that are common to most digital designs. The error detection and correction, RAM control and address generation, and parallel-to-serial conversion features of the T1 coprocessor are also widely used in system design. Furthermore, digital data communication of both voice and data plays an increasingly important role in modern systems.

Although this design example shows a T1 serial transmission application operating at 1.544 MHz, the same concept can be applied to create high-performance subsystems for applications such as local area networks or disk controllers. The EPM5130 EPLD supports serial data rates of up to 50 megabits per second.

4.6.8 Controlling Complex CCD Imaging Systems with the EPS464 EPLD

Advances in VLSI-scale microelectronics have allowed manufacturers of video and optical equipment to replace optical film storage media with arrays of light-sensitive electronic imaging elements. These video systems use analog charge-coupled device (CCD) technology to retain image data, as well as a high-speed digital control system to convert the image into a standard video format.

The Altera EPS464 Synchronous Timing Generator (STG) is an ideal device for implementing the necessary digital control logic. The STG is well suited for generating complex waveforms, control signals, and state machines. The EPS464 EPLD contains 64 macrocells optimized for timing and waveform synthesis applications. Thirty-two of these macrocells are connected to I/O pins, while the other 32 are available for buried logic such as state machine registers and user-defined n-bit counters. A global bus feeds all macrocells within the device allowing system speeds of 66 MHz.

The complex digital control system for a hand-held video camera requires a combination of high speed and high density. Not only must the system retrieve image data from the imaging element, it must also process the analog data into a standard format for broadcast or storage on magnetic tape. All of the functional blocks must work together and be perfectly synchronized.

Most hand-held cameras use a CCD element as the image capture medium. The CCD array does an excellent job of digitizing the incident images at high speed; however, the larger the resolution of the array, the faster the digital control system must operate. An array with a resolution of 910 x 525 pixels generates over 450,000 data points for every image. To meet the NTSC standard of 30 images per second, the control system must operate at 14.318 MHz. The combination of high density, high performance and architectural flexibility of the EPS464 is an ideal solution to this problem.

Figure 4.6.8.1 shows a typical implementation of a controller for a hand-held 8-mm video camera. The application uses a 910 x 525-bit CCD imaging array and circuitry to format the image data into an NTSC video standard.

The design contains three main blocks: a horizontal (X) counter, a vertical (Y) counter, and a synchronous signal generator (SSG). In Figure 4.6.8.1 the two counters are shown as timing generators which are used to format the video display into pixels per line and lines per screen. The SSG uses these counter values to generate the necessary control patterns and formatting signals.

The size of the counters varies, according to the desired video format and the resolution of the display. The NTSC format specifies an interlaced display scheme; therefore, the system must count to one-half of the 525 lines in the display, or 262.5 lines. However, the count is difficult to control since it depends on half clock cycles. To solve this problem, the screen is divided into a series of 1050 half-lines, with each frame using 525. By offsetting the display by half a horizontal line and displaying the two "half-pictures" in rapid succession, the interlace appears fluid, and the result is a very good representation of the original picture. This approach takes advantage of the physiology of human vision: although the human eye is good at distinguishing a wide range of colors, it is not so good at distinguishing small or rapid movements. Consequently, an image updated 30 times per second appears to have fluid motion.

Given these system requirements, the horizontal counter needs 10 bits to count the

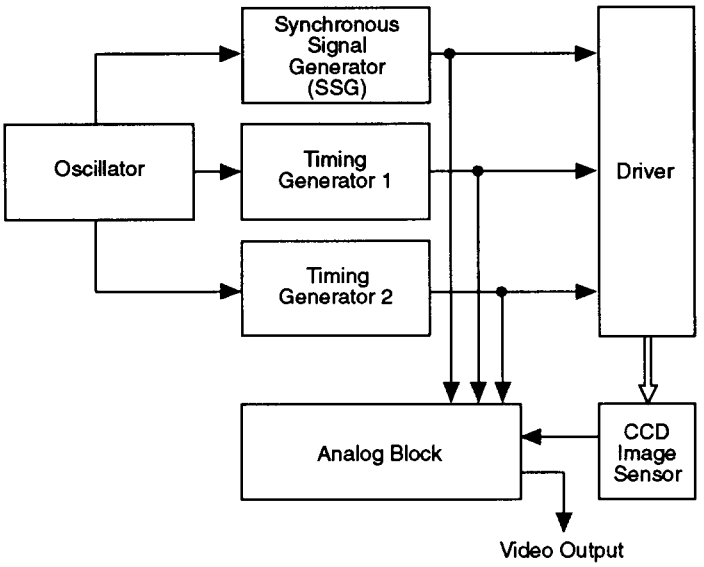


Figure 4.6.8.1 Typical CCD Controller

910 pixels per line and the vertical counter needs 11 bits to count the 1050 lines per screen. By using the 1050 half-line approach, the counters will use integers rather than a cumbersome scheme of half clock cycles.

The NTSC standard has three key specifications: lines per screen, line period, and screens per second. Since each image must be synchronized by the display hardware, and any variation from the standard would cause the image to roll on the screen, the lines-per-screen specification is fixed at 525. However, not all 525 lines are actually shown on the display. Twenty are used for formatting and equalization intervals, while the remaining 505 are dedicated to image data.

The second specification is the amount of time that is permitted to display a line. Each line is encoded during a 63.556s period established by the incoming video signal. To encode the 910 horizontal pixels in the application, the control system must divide each horizontal time period into 910 individual segments and sample the analog video signal at each of those points. Each sample corresponds to the clocking out of the next word of analog data from the CCD array. With a horizontal time period of 63.556s, the sampling rate is 14.318 MHz.

The third NTSC specification requires that an image must be updated 30 times per second, requiring a refresh every 33.33 ms. Since the NTSC is an interlaced format with two frames per image, the actual frame refresh rate is 60 Hz. This rate is easily achieved with the EPS464 EPLD as the digital control system.

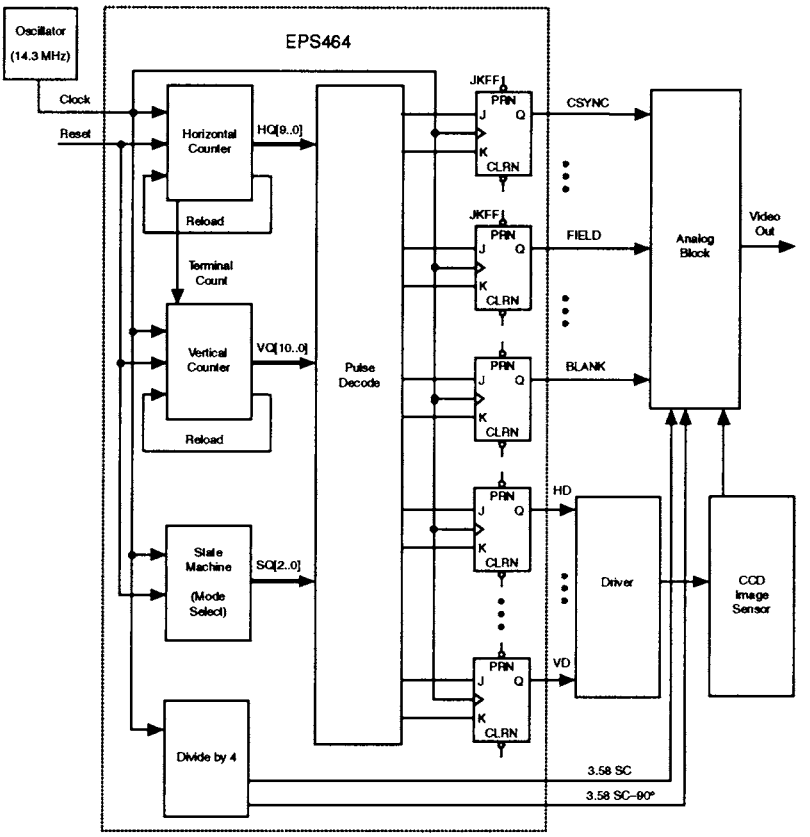


Figure 4.6.8.2 Top Level of the CCD Imaging System Control Application

The final video output is an analog signal that combines analog voltages representing intensity and color information with digital formatting pulses. Although digital and analog levels are combined with a special-purpose analog mixing device, mixing is actually performed under the control of the digital system. The result is a single composite video signal for broadcast or storage on tape.

Figure 4.6.8.2 shows this application implemented in the Altera EPS464 EPLD. All five subfunctions (horizontal counter, vertical counter, state machine, pulse decode module, and clock divider) fit inside a single EPS464 and can be easily modified to meet the requirements of any application.

4.7. References

Ahanin, B. and Lytle, C., "Methods and Apparatus for Facilitating Scan Testing of Asynchronous Logic Circuitry," U.S. Patent Application (pending).

Altera Application Brief AB 76, "Using Expanders to Build Registered Logic in EPM5000-Series MAX EPLDs," October 1990, Version 2.

Altera Application Note AN 22, "Designing with AHDL," Oct. 1990, Version 1.

Altera, "Altera EDIF Writer Specification Version 3.0," Specification Number 16C-01199, Dec. 1991.

Altera, "Valid/Cadence Design Kit for PC Users," Altera Software Utility.

Altera, "Viewlogic Design Kit for PC Users," Altera Software Utility.

ANSI/EIA - 548-1988, "Electronic Design Interchange Format Version 2 0 0."

Birkner, J. et al. (editor), *Programmable Array Logic Handbook, First Edition*, Monolithic Memories Inc., 1978.

Datta, S. and Kung, E., "EDIF: Its Contribution to the PLD Design Flow," *The PLD Design Conference and Exhibit Proceedings*, 1991.

Fiduccia, C.M. and Mattheyses, R.M., "A Linear-Time Heuristic for Improving Network Partitions," *19th Design Automation Conference*, Paper 13.1, pp 175-181.

Hartmann, R., "Estimating Gate Complexity of Programmable Logic Devices," *VLSI Design Magazine*, pp 100-102, May 1984.

Kernighan, B.W. and Lin, S., "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell System Technical Journal*, Feb. 1970, pp 291-307.

Kitson, B. et al. (editor), *Programmable Logic Array Handbook*, Advanced Micro Devices Inc., 1984, pp. 2-29 through 2-40.

Munoz, R. R. et al., "Automatic Partitioning of Programmable Logic Devices," *VLSI Systems Design*, Oct. 1987, pp. 74-76, 78 and 86.

Wolf, Stanley, *Silicon Processing for the VLSI Era*, Lattice Press, CA, 1990, pp 623-635.

Wong et al., "Programmable Logic Device with Array Blocks Connected Via Programmable Interconnect," U.S. Patent 4,871,930, Oct. 3, 1989, Col 2, line 43 et seq.

Index

A

ABEL 143
Act1 110–113
Act2 113–117
Act3 117–118
ACTMAP 140
ADL 143
Ahanin 202
Ahrens 97, 118
Allen 97, 99
Altera 191, 216, 235
amorphous silicon 100
analog FPGA 4
ANSI 205
antifuse 99
    dielectric 100
    F antifuse 109
    ONO 100
    resistance 100
Apple 82
array-based architecture 184
automatic test vector generation 202

B

binary counter 71
binary decision diagram 129
Birkner 100, 171
bit-per-state 160
    see also one-hot 69
bit-slice 49
Blkmake 57, 59
Boolean equations 128
boundary scan 45, 80
Brayton 129
Breuer 58
Britton 15
buffering 25

C

capacity 51, 124
carry chain 44, 219
carry hard macro 156
carry logic 44, 70
carry-propagation 153
carry-select 154, 160
Carter 15, 97, 99
cascade 26
cascade chain 219
channel density 104
Chen 101
Chene 57
Chiang 101, 102, 103
Chortle 55, 57
Chow 3, 15
cip, see pip 20
CLB 19, 22, 29, 36, 44
    see also logic module 113
    see also macrocell 180
CLBMAP 64

Cliff 15
clock routing 33, 49
C-module 113, 117
CMOS 18
Compiler 208
Complex PLD 3
configurable display device 82
configurable interconnect point, see pip 20
configurable logic block, see CLB 19
configurable memory 45
configuration
  see programming 52
configuration bitstream 22
configuration memory 15
Cong 55
controllability 52
counter 70, 144, 226
Cox 75
CPLD 3, 171
crossbar switch 22

D

Datta 205
debugging 84, 131
density 10, 23
design migration 11, 200
digital signal processing 77, 164
direct addressing 120
direct connect 26, 71
  see also cascade 219
direct connection 32
distributed arithmetic 77
DMA controller 85
Donath 29
double-entry symmetry 116
double-length lines 46
dual feedback 182, 185
Dunlop 58
duplicated logic 57, 69

E

Ebeling 3, 15
EDIF 211
ElAyat 97
electronic scaffolding 84
ElGamal 29, 97, 106, 107, 109, 111, 118
EP1810 184–187
EPROM 173
expander 189, 190, 198, 229
expander product term 189

F

fan-in 134
fan-out 133
FastTrack 219, 222
Fiduccia 58, 216
Field Programmable Gate Array, see FPGA
Field-Programmable Analog Array (FPAA) 4
fine-grain vs. coarse grain 27
fitting 172, 204, 216
  see also partitioning 54
flexible processor 90
flip-flop 28, 69
floppy disk 82
force-directed relaxation 60
FPGA
  advantages 4
  architecture 3
  cost comparison with MPGA 9
  definition 2
  density 8
  design migration to MPGA 11
  disadvantages 8
  speed 9, 10
  taxonomy 3
Francis 55, 57
Frankle 62
function generator, see lookup table 20

Furtek 15

G

Ganglion 75
gate array 1
general-purpose interconnect 46
Gerzberg 97, 100
global clock 70
global interconnect 33, 48
globally connected 184
gobbler 138
Goto 60
Graham 100, 118
graphic editor 208
Greene 26, 107
grow back 99

H

Hamdy 97, 100
hard macro 137
Hartmann 171
Hartoog 58
Hashimoto 104
Hastie 15
Hauck 15
Heller 29, 49
high-performance counter 150
Hill 15, 29
Holley 67
Holmberg 100
Hsieh 15, 43, 97

I

I/O block 182
if-then-else DAG 129
in-circuit verification 6, 54
incremental placement 67, 130
incremental routing 67
input segment 108
IOB 30, 37, 45
    see I/O block 22
iterative design 70

J

JEDEC 216
jitter-bounded PLL 165
JTAG, see boundary scan 45

K

Karplus 55, 129
Kawana 15
Kean 15
Kernighan 216
Kirkpatrick 60
Kitson 190

L

LAB 188, 197, 221
    see also CLB 22
    see also logic module 113
leakage current 100
Lee 4
left edge algorithm 104
levels of logic 133
library 54, 66
Library Mapping Files 211
life cycle 7
Lim 100
limit-bumping algorithm 62
Lisa 4
loadable counter 144
logic array 180
logic array block 188
    see LAB 187
logic module 113
    see also CLB 22
    see also macrocell 180
logic optimization 66, 213
logic synthesis 63, 172, 204
Logic Synthesizer 213
long line 32
lookup table 20, 69, 219

Lorenzetti 104
LPM 67
LSI 130
LUT 20
Lyons 17

M

macro, combinable 159
macrocell 180, 186, 189, 197
see also CLB 22
macrofunction 179
macrofunction library 211
magic box
see switchbox 22
Mailhot 129
mapping 129
see fitting 216
see partitioning 54
Marr 106
Mask Programmed Gate Array (MPGA) 1
Mask-Programmed Logic Device 200
mask-programmed logic device (MPLD) 200
Maunder 80
MAX5000 187–195
MAX7000 195–200
memory 45
memory cell in SRAM FPGA 15
message processor 207
Michael 0
microparameter 224
min-cut 58, 216
minimization 204
Mintzer 77
MIS-II 55
MIS-pga 55
module utilization 124
MPGA 1, 29
multi-chip partitioning 206
multi-gate macros 134
multiplexer 21
multiplication 75, 77
Munoz 190
Murgai 55, 129
Muroga 15

N

nearest-neighbor connections
see direct connect 26
neural network 75
Nilsson 61
NRE 5, 7

O

observability 52
one-hot 69
see also bit-per-state 160
output enable 142
output pin swapping 34
output segment 109

P

pad limited 10
PAL 184
PALASM 140, 143
Pamela 2
parallel logic expander 198
partitioner 216
partitioning 54, 216
see also fitting 216
pass transistor 21
Penfield 62
performance 127
Perle-0 88
Perle-1 88
PIA 189, 193
pin assignment 130
pin swapping 34, 61
pip 20, 25, 26
pipelining 70
placement 41, 59, 67, 130
directional bias 60
incremental 68

PLD 29
PLICE 100
position tracker 84
power consumption 19
Preas 109
pre-scaled counter 150
probe points 65
product term 177
program 22
programmability
    effect on density 8
    effect on speed 9
Programmable Array Logic (PAL) 1
programmable interconnect array 189, 192, 199
programmable interconnect point, see pip 20
Programmable Logic Device (PLD) 1
Programmable Multi-level Devices (PMD) 3
Programmer Object Files 216
programming 52, 118, 131, 173
    overhead 8, 10
    re-programming 52
    time 131
programming yield 6, 18, 173, 183
prototype 67, 84, 87
pulse-steal PLL 164

Q

Quickturn Systems 87

R

RAM 45
readback 52, 90
reconfiguration 52
    see reprogramming 52, 79
Report File 212
repowering buffers 32, 40, 49
reprogramming 17, 70, 76, 79–83
resource limits 29
ripple-carry adder 71
Roesner 100
Rose 27, 28
Rosendahl 80
routability 25, 29
routing 61, 67, 104, 130
    global route 61
    maze route 61
Rubenstein 62
Rudell 129

S

Sawkar 55
scan-testing 201
schematic editor 53
Schlageter 97
Sechen 59, 60
security 53, 182
segment 22, 32
segmented routing channel 26, 106
series pass transistor addressing 120
74AS161 138
shared logic expander 197, 198
simulated annealing 60
simulation 67
Simulator Netlist File 201
slack 62
sloppy design 9
Smith 97, 99
S-module 113, 117
    combinable 117
sneak path 123
soft macro 135, 137
Soukup 61
speed 24, 43, 51
speed binning 7
Splash 89
state-machine generator 53
Steiner tree 109
Stopper 100
sub-families 49
sum-of-products 177

switchbox 22
Symbol Editor 208
synchronous binary counter 144
synthesis 53, 67, 141
    technology specific 63

T

TA161 138
table lookup
    see lookup table 20
Tan-Nguyen 82
tape drive 82
technology mapping
    see partitioning 54
telecommunications 164
testing 6, 18, 79, 118
three-state 141
three-state buffers 39
    placement constraint 59
    placement constraints 60
threshold drop 21
time-dependent dielectric breakdown 102
timing-driven placement 130
TLU
    see lookup table 20
tooling 4
Trimberger 43, 57
two-segment routing 107

V

video controller 83
virtual hardware 90
volatility 17

W

Walters 165
waveform editor 207, 209
Whitney 97
Whitten 97, 100
wiring channel 22
Wolf 173
Wong 97, 106, 184
Woo 55

X

XACT Design Editor 64
X-BLOX 63
XC2000 29–35, 40
XC3000 35–43
XC4000 43–51
XC4000A 49
Xilinx 17
Xnfmap 56

Appendix D.1

OCLC FirstSearch: Detailed Record

Your requested information from your library LOS ANGELES PUB LIBR

Return

WorldCat results for: **au: bhat and ((ti: Novel and ti: techniques and ti: high and ti: performance and ti: field and ti: programmable and ti: logic and ti: devices))**. Record **4** of **6**.

Detailed Record | Add/View Comments

Novel techniques for high performance field programmable logic devices /

Narasimha B **Bhat**

1993
English Book : Thesis/dissertation/manuscript Archival Material xiii, 209 leaves : illustrations ; 28 cm

GET THIS ITEM

Availability: **Check the catalogs in your library.**
- Libraries worldwide that own item: 1
- Connect to the catalog at your library

External Resources:
- Cite This Item

Title: **Novel techniques for high performance field programmable logic devices /**

Author(s): Bhat, Narasimha B.

Year: 1993

Description: xiii, 209 leaves : illustrations ; 28 cm

Dissertation: Ph. D. in Electrical Engineering and Computer Sciences; University of California, Berkeley; 1993

Language: English

SUBJECT(S)

Named Corp: University of California, Berkeley. Department of Electrical Engineering and Computer Sciences.

Genre/Form: Academic Dissertation
Academic theses.
Academic theses.
Thèses et écrits académiques.

Note(s): Includes bibliographical references.

General Info: **Other format available:** Online version:; Bhat, Narasimha B.; Novel techniques for high performance field programmable logic devices.; 1993

Class Descriptors: **LC:** T7.6.1993

Responsibility: by Narasimha B. Bhat.

Material Type: Thesis/dissertation (deg); Manuscript (mss)

Document Type: Book; Archival Material

Date of Entry: 19940906

Update: 20220408

Accession No: **OCLC:** 215199002

Database: WorldCat

FirstSearch® Copyright © 1992-2023 OCLC as to electronic presentation and platform. All Rights Reserved.
E-mail address: FirstSearch@oclc.org **Location:** https://FirstSearch.oclc.org

Return

Appendix D.2

Berkeley Library
UNIVERSITY OF CALIFORNIA
DIGITAL COLLECTIONS

Preview

Select file:

ERL-93-80

Copyright © 1993, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Description

Field programmable logic devices (FPLDs) are fast emerging as viable alternatives to mask programmed parts because of their rapid time-to-market and low costs. Their application has, however, been limited to implementing random logic, with non-critical timing specifications. This work attempts to advance FPLD usage to high-performance applications as well. A two-pronged approach is adopted. In the first part, CAD algorithms aimed at improving routability and performance of designs mapped to existing look-up table (LUT) based field programmable gate arrays (FPGAs) are developed. The concept of a two-input LUT primitive cell is introduced. This reduces the number of library patterns that require to be stored for an LUT library, and makes it feasible to extend performance-driven library based technology mapping techniques to LUT FPGAs. The performance-driven mapping algorithm accounts for interconnect delay and provides area-delay trade offs. Experiments on benchmark designs show the effectiveness of the new algorithms. In the second part, a new FPLD architecture is introduced. The architecture is based on the concept of time-sharing of logic and routing resources in an effort to have a fully routable, CAD friendly FPLD with predictable timing performance and efficient silicon usage. Real-time reconfiguration of logic and routing resources implements a given circuit in a folded pipe-line fashion, pipe lining at the gate level. Several possible variations of the basic architecture are discussed. A simple synthesis scheme is developed and experimental results are reported. Area and timing analyses demonstrates advantages over existing FPGAs.

Details

Title
Novel Techniques for High Performance Field Programmable Logic Devices

Creator
Computer Science Division, Publisher (https://digicoll.lib.berkeley.edu/search?f1=creator&as=1&sf=title&so=a&rm=&m1=e&p1=Computer Science Division&ln=en)
Bhat, Narasimha B., Author (https://digicoll.lib.berkeley.edu/search?f1=creator&as=1&sf=title&so=a&rm=&m1=e&p1=Bhat, Narasimha B.&ln=en)

Published
1993-11-01

Full Collection Name
Electrical Engineering & Computer Sciences Technical Reports

Other Identifiers
ERL-93-80

Type
Text

Format
technical reports (https://digicoll.lib.berkeley.edu/search?f1=subject&as=1&sf=title&so=a&rm=&m1=p&p1=technical reports&ln=en)

Extent
232 p

Archive
The Engineering Library

Usage Statement
Researchers may make free and open use of the UC Berkeley Library's digitized public domain materials. However, some materials in our online collections may be protected by U.S. copyright law (Title 17, U.S.C.). Use or reproduction of materials protected by copyright beyond that allowed by fair use (Title 17, U.S.C. § 107) requires permission from the copyright owners. The use or reproduction of some materials may also be restricted by terms of University of California gift or purchase agreements, privacy and publicity rights, or trademark law. Responsibility for determining rights status and permissibility of any use or reproduction rests exclusively with the researcher. To learn more or make inquiries, please see our permissions policies (https://www.lib.berkeley.edu/about/permissions-policies).

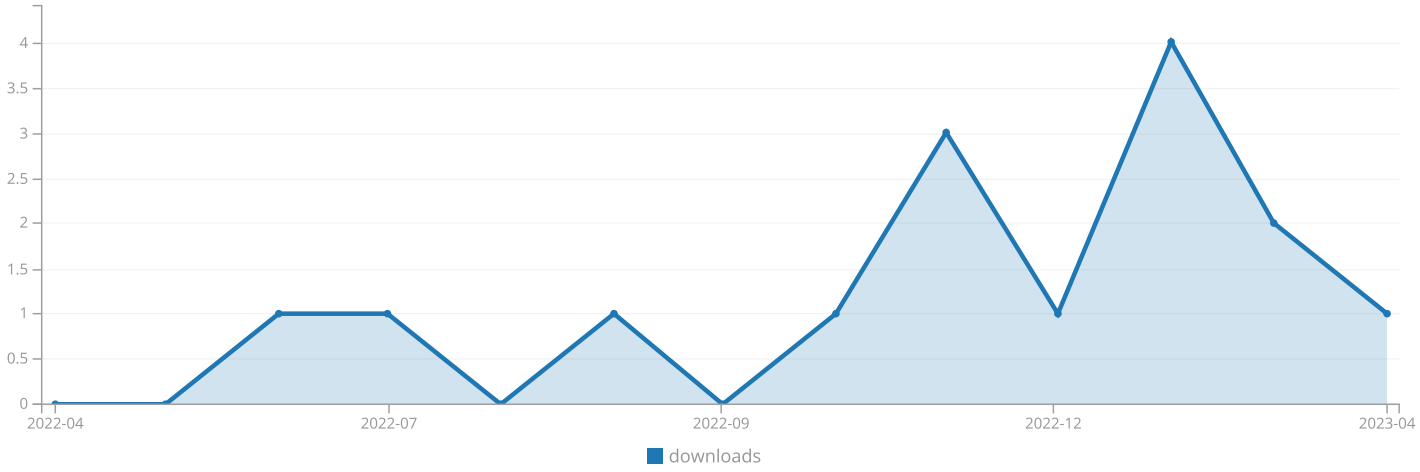
Collection
EECS Technical Reports (https://digicoll.lib.berkeley.edu/search?&sf=title&so=a&rm=&p=collection%3A%27EECS%20Technical%20Reports%27&ln=en)

Files

Action	Filename	Size	Description
Download	ERL-93-80.pdf	11.5 MB	-

Statistics

Downloads
from 2022-04-14
to 2023-04-14
By Months
Update
Export



Download Full History (/tindstats/bibdoc_downloads_dl?recid=137690&download=csv)

Download (https://digicoll.lib.berkeley.edu/record/137690/files/ERL-93-80.pdf)

Formats

Add to Basket (/yourbaskets/add?recid=137690)

Appendix D.3

Novel Techniques for High Performance Field-Programmable Logic Devices

Narasimha B. Bhat

EECS Department
University of California, Berkeley
Technical Report No. UCB/ERL M93/80
November 1993

http://www2.eecs.berkeley.edu/Pubs/TechRpts/1993/ERL-93-80.pdf (http://www2.eecs.berkeley.edu/Pubs/TechRpts/1993/ERL-93-80.pdf)

Field programmable logic devices (FPLDs) are fast emerging as viable alternatives to mask programmed parts because of their rapid time-to-market and low costs. Their application has, however, been limited to implementing random logic, with non-critical timing specifications. This work attempts to advance FPLD usage to high-performance applications as well. A two-pronged approach is adopted. In the first part, CAD algorithms aimed at improving routability and performance of designs mapped to existing look-up table (LUT) based field programmable gate arrays (FPGAs) are developed. The concept of a two-input LUT primitive cell is introduced. This reduces the number of library patterns that require to be stored for an LUT library, and makes it feasible to extend performance-driven library based technology mapping techniques to LUT FPGAs. The performance- driven mapping algorithm accounts for interconnect delay and provides area-delay trade offs. Experiments on benchmark designs show the effectiveness of the new algorithms. In the second part, a new FPLD architecture is introduced. The architecture is based on the concept of time-sharing of logic and routing resources in an effort to have a fully routable, CAD friendly FPLD with predictable timing performance and efficient silicon usage. Real-time reconfiguration of logic and routing resources implements a given circuit in a folded pipe-line fashion, pipe lining at the gate level. Several possible variations of the basic architecture are discussed. A simple synthesis scheme is developed and experimental results are reported. Area and timing analyses demonstrates advantages over existing FPGAs.

Advisor: Ernest S. Kuh

BibTeX citation:

```
@phdthesis{Bhat:M93/80,
    Author = {Bhat, Narasimha B.},
    Title = {Novel Techniques for High Performance Field-Programmable Logic Devices},
    School = {EECS Department, University of California, Berkeley},
    Year = {1993},
    Month = {Nov},
    URL = {http://www2.eecs.berkeley.edu/Pubs/TechRpts/1993/2452.html},
    Number = {UCB/ERL M93/80},
    Abstract = {Field programmable logic devices (FPLDs) are fast emerging as viable alternat
}
```

EndNote citation:

```
%0 Thesis
%A Bhat, Narasimha B.
%T Novel Techniques for High Performance Field-Programmable Logic Devices
%I EECS Department, University of California, Berkeley
%D 1993
%@ UCB/ERL M93/80
%U http://www2.eecs.berkeley.edu/Pubs/TechRpts/1993/2452.html
%F Bhat:M93/80
```

(https://eecs.berkeley.edu/)

About (https://eecs.berkeley.edu/about)

History (https://eecs.berkeley.edu/about/history)

Diversity (https://eecs.berkeley.edu/about/diversity)

Visiting (https://eecs.berkeley.edu/about/visiting)

Special Events (https://eecs.berkeley.edu/about/special-events)

Academics (https://eecs.berkeley.edu/academics)

Undergrad Admissions & Programs (https://eecs.berkeley.edu/academics/undergraduate)

Graduate Admissions & Programs (https://eecs.berkeley.edu/academics/graduate)

Courses (https://eecs.berkeley.edu/academics/courses)

Prospective Women Students (https://eecs.berkeley.edu/academics/prospective-women)

Current Students (https://eecs.berkeley.edu/resources/students)

Research (https://eecs.berkeley.edu/research)

Areas (https://www2.eecs.berkeley.edu/Research/Areas/)

Centers & Labs (https://www2.eecs.berkeley.edu/Research/Areas/Centers/)

Projects (https://www2.eecs.berkeley.edu/Research/Projects/)

Technical Reports (https://www2.eecs.berkeley.edu/Pubs/TechRpts/)

PhD Dissertations (https://www2.eecs.berkeley.edu/Pubs/Dissertations/)

Joint Colloquium (https://eecs.berkeley.edu/research/colloquium)

People (https://eecs.berkeley.edu/people)

Directory (https://www2.eecs.berkeley.edu/Directories/directory-nostudents.html)

Leadership (https://eecs.berkeley.edu/people/leadership)

Faculty (https://eecs.berkeley.edu/people/faculty)

Students (https://eecs.berkeley.edu/people/students)

Staff (https://eecs.berkeley.edu/people/staff)

Alumni (https://eecs.berkeley.edu/people/alumni)

Resources (https://eecs.berkeley.edu/resources)

Room Reservations (https://eecs.berkeley.edu/resources/facilities/room-reservations)

My EECS Info (https://www2.eecs.berkeley.edu/deptinfo)

For Students (https://eecs.berkeley.edu/resources/students)

For Grads (https://eecs.berkeley.edu/resources/grads)

For Undergrads (https://eecs.berkeley.edu/resources/undergrads)

GSIs/Readers/Tutors (https://eecs.berkeley.edu/resources/gsis)

IT Services (http://iris.eecs.berkeley.edu/)

Facilities/Safety (https://eecs.berkeley.edu/resources/facilities)

For Faculty/Staff (https://eecs.berkeley.edu/resources/faculty-staff)

Visiting Scholars (https://eecs.berkeley.edu/resources/visiting-scholars)

Industry (https://eecs.berkeley.edu/industry)

Recruit Students (https://eecs.berkeley.edu/industry/recruit-students)

Entrepreneurial Activity (https://eecs.berkeley.edu/industry/entrepreneurship)

Connect (https://eecs.berkeley.edu/connect)

Support Us (https://eecs.berkeley.edu/connect/support)

K-12 Outreach (https://eecs.berkeley.edu/connect/k-12)

Contact (https://eecs.berkeley.edu/contact)

Faculty Positions (https://eecs.berkeley.edu/faculty-jobs)

(https://eecs.berkeley.edu/)

EE (https://ee.berkeley.edu/)

CS (https://cs.berkeley.edu/)

UC Berkeley (http://berkeley.edu/)

Berkeley Engineering (http://engineering.berkeley.edu/)

News (https://eecs.berkeley.edu/news)

Events (https://eecs.berkeley.edu/events)

(https://twitter.com/Berkeley_EECS)

(https://www.instagram.com/ucb_eecs/)

(https://www.linkedin.com/school/uc-berkeley-eecs/)

(http://www.youtube.com/c/UCBerkeleyEECSEvents)

© 2020 UC Regents

- Privacy Policy (https://eecs.berkeley.edu/privacy-policy)

Appendix D.4

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9430394

Novel techniques for high performance field programmable logic devices

Bhat, Narasimha B., Ph.D.

University of California, Berkeley, 1993

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

Novel Techniques for High Performance Field Programmable Logic Devices

by

Narasimha B. Bhat

B.E. (Mangalore University, India) 1988
M.E. (Indian Institute of Science, Bangalore, India) 1989

A dissertation submitted in partial satisfaction of the
requirements for the degree of

Doctor of Philosophy

in

Engineering-Electrical Engineering
and Computer Sciences

in the

GRADUATE DIVISION

of the

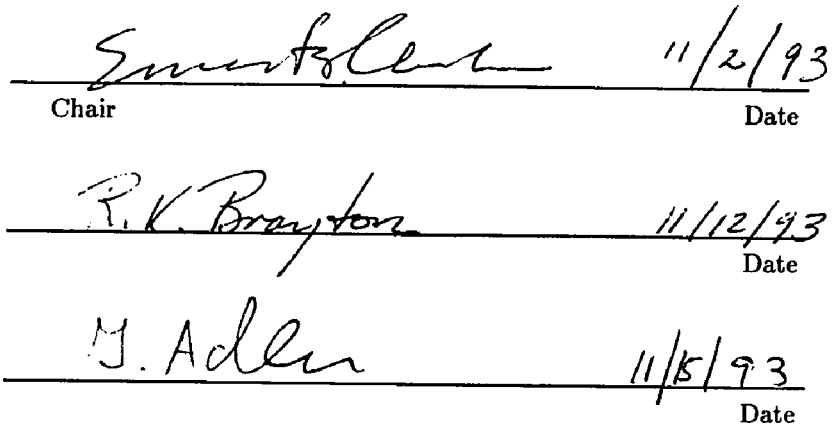
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Ernest S. Kuh, Chair
Professor Robert K. Brayton
Professor Ilan Adler

1993

The dissertation of Narasimha B. Bhat is approved:



Chair Date

Date

Date

University of California at Berkeley

1993

Abstract

Novel Techniques for High Performance Field Programmable Logic Devices

by

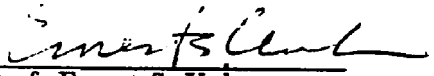
Narasimha B. Bhat

Doctor of Philosophy in Engineering-Electrical Engineering and Computer Sciences

University of California at Berkeley

Professor Ernest S. Kuh, Chair

Field programmable logic devices (FPLDs) are fast emerging as viable alternatives to mask programmed parts because of their rapid time-to-market and low costs. Their application has, however, been limited to implementing random logic, with non-critical timing specifications. This work attempts to advance FPLD usage to high-performance applications as well. A two-pronged approach is adopted. In the first part, CAD algorithms aimed at improving routability and performance of designs mapped to existing look-up table (LUT) based field programmable gate arrays (FPGAs) are developed. The concept of a two-input LUT primitive cell is introduced. This reduces the number of library patterns that require to be stored for an LUT library, and makes it feasible to extend performance-driven library-based technology mapping techniques to LUT FPGAs. The performance-driven mapping algorithm accounts for interconnect delay and provides area-delay trade-offs. Experiments on benchmark designs show the effectiveness of the new algorithms. In the second part, a new FPLD architecture is introduced. The architecture is based on the concept of time-sharing of logic and routing resources in an effort to have a fully routable, CAD friendly FPLD with predictable timing performance and efficient silicon usage. Real-time reconfiguration of logic and routing resources implements a given circuit in a folded pipe-line fashion, pipe-lining at the gate level. Several possible variations of the basic architecture are discussed. A simple synthesis scheme is developed and experimental results are reported. Area and timing analyses demonstrate advantages over existing FPGAs.



Prof. Ernest S. Kuh
Thesis Committee Chairman

Novel Techniques for High Performance Field Programmable Logic Devices

Copyright © 1993

Narasimha B. Bhat

Contents

Table of Contents iii

List of Figures viii

List of Tables xi

Acknowledgements xii

1 Introduction and Overview 1
- 1.1 FPLD classification . . . 3
- 1.2 CAD for FPLDs . . . 5
- 1.3 Programmable logic system . . . 9
- 1.4 FPLD advantages . . . 10
- 1.5 Focus . . . 10
- 1.6 Thesis overview . . . 12

2 Terminology 14

I Performance-oriented Synthesis 21

3 Overview 22
- 3.1 Performance-oriented mapping . . . 23
- 3.2 Routable mapping . . . 24
- 3.3 LUTs and logic modules . . . 25

4 Routable Technology Mapping 26
- 4.1 Motivation . . . 26
  - 4.1.1 Cause of incomplete routing . . . 27
  - 4.1.2 Problem with existing CAD tools . . . 27
  - 4.1.3 Our strategy . . . 28
- 4.2 Terminology . . . 31
- 4.3 FPGA Model . . . 32
  - 4.3.1 FPGA model . . . 32

4.3.2 qblock . . . 33
4.3.3 Look-up table . . . 35
4.3.4 Routing resources . . . 35
4.4 Overview of our approach . . . 35
4.5 RFR algorithm . . . 36
4.5.1 Decomposition into IOLs . . . 37
4.5.2 Global routing . . . 37
4.5.3 Checking logic feasibility . . . 38
4.5.4 Simulated annealing . . . 39
4.5.5 Re-synthesizing a packing solution . . . 40
4.6 Experimental results . . . 40
4.7 Conclusions . . . 45

5 Performance-oriented Mapping **47**
5.1 Motivation . . . 47
5.2 Terminology . . . 51
5.3 Review of library-based mapping . . . 51
5.3.1 Tree covering algorithm . . . 52
5.3.2 Partitioning DAGs . . . 55
5.4 Why has this not been done before? . . . 55
5.5 TIL based LUT library . . . 56
5.5.1 TILs as primitive cells . . . 56
5.5.2 Small library . . . 57
5.6 **dpmap** Algorithm . . . 59
5.6.1 Input network → TINN . . . 59
5.6.2 Tree covering . . . 61
5.6.3 Handling reconvergent fanout . . . 61
5.7 Fast tree matching . . . 63
5.7.1 Principle . . . 64
5.7.2 Example . . . 65
5.7.3 Drawbacks . . . 68
5.8 Experimental results . . . 70
5.8.1 Performance-driven mapping . . . 72
5.8.2 Area-delay trade-off . . . 72
5.8.3 Minimum depth . . . 72
5.8.4 Minimum area . . . 75
5.9 Conclusions . . . 76

II Performance-oriented Architecture **77**

6 Overview **78**

7 Why Another Architecture? **80**
7.1 Routing issues . . . 81
7.1.1 Architecture approach . . . 81
7.1.2 CAD approach . . . 82
7.1.3 Summary . . . 85
7.2 Timing issues . . . 85
7.2.1 Dominant interconnect delay . . . 85
7.2.2 Widely varying interconnect delay . . . 87
7.3 CAD issues . . . 87
7.4 Desired features . . . 88
7.5 New architecture . . . 89

8 New Architecture **90**
8.1 Principle of levelize and hold . . . 90
8.1.1 Signal propagation in logic circuits . . . 90
8.1.2 Levelize-and-hold . . . 94
8.1.3 Sequential elements . . . 95
8.2 *Dharma* Architecture . . . 95
8.2.1 High-level block diagram . . . 95
8.2.2 Detailed block diagram . . . 98
8.2.3 Clocking scheme . . . 100
8.2.4 Multi-chip operation . . . 101
8.2.5 Dynamic logic modules . . . 103
8.2.6 Pass-buffers . . . 104
8.2.7 Dynamic interconnection array . . . 105
8.2.8 Latches . . . 107
8.2.9 Level generation circuitry . . . 107
8.3 Operating procedure . . . 110

9 Examples to Illustrate *Dharma* **114**
9.1 *Simple Dharma* . . . 114
9.2 Combinational example 1 . . . 115
9.3 Combinational example 2 . . . 118
9.4 Sequential example . . . 122

10 Variations and Modifications **134**
10.1 Dynamic logic module . . . 134
10.1.1 Complex LUT DLMs . . . 135
10.1.2 Multiplexer based DLMs . . . 136
10.1.3 PLA based DLMs . . . 136
10.1.4 Heterogeneous DLM array . . . 136
10.1.5 Technology dependent modifications . . . 138
10.2 Dynamic interconnect array . . . 138
10.2.1 Shift register at crosspoint . . . 138
10.2.2 Multiplexer based crossbar . . . 140

10.2.3 Partially hard-wired DIA . . . 141
10.2.4 Binomial concentrator based DIA . . . 141
10.2.5 DIA as a *Copier* . . . 144
10.2.6 Sparsely connected K-class DIA . . . 145
10.2.7 Functional DIA . . . 148
10.2.8 Technology dependent modifications . . . 149
10.3 Level expander . . . 149
10.3.1 Example . . . 149
10.3.2 Expander circuitry . . . 151
10.3.3 Local level generator . . . 153

11 Synthesis for *Dharma* **155**
11.1 Introduction . . . 155
11.1.1 Topological levelization . . . 156
11.1.2 Temporal partitioning . . . 158
11.1.3 Handling other designs . . . 159
11.2 Levelization as an integer linear program . . . 159
11.2.1 Upper and lower level limits . . . 160
11.2.2 Level variable . . . 160
11.2.3 Level of a node . . . 160
11.2.4 Fanin constraints . . . 161
11.2.5 Nodes per level . . . 161
11.2.6 Complete ILP . . . 161
11.3 Temporal partitioning . . . 162
11.3.1 Uniform DLM distribution . . . 162
11.3.2 Modified Kernighan-Lin heuristic . . . 163
11.3.3 Move selection . . . 165
11.4 Experimental results . . . 166
11.5 Conclusions . . . 168

12 Architecture Analysis **171**
12.1 Area analysis . . . 171
12.1.1 Assumptions . . . 171
12.1.2 Area components of *Dharma* . . . 172
12.1.3 The analysis . . . 173
12.1.4 Comparison with Xilinx 2000 series . . . 175
12.1.5 Comparison with Xilinx 3000 series . . . 177
12.1.6 Comparison with Xilinx 4000 series . . . 180
12.1.7 Comparison with the AT&T ORCA . . . 180
12.2 Timing analysis . . . 181
12.2.1 Notation . . . 181
12.2.2 EFP timing equation . . . 182
12.2.3 S timing equation . . . 184
12.2.4 Remarks . . . 186
12.3 Experimental verification . . . 187

12.4 Conclusions 189

13 Unexplored Terrain **191**
13.1 Logic synthesis for *Dharma* 191
13.2 Repetitive structures 195
13.2.1 Principle 195
13.2.2 Example 195
13.2.3 Remarks 197

14 Conclusions **200**

Bibliography **202**

List of Figures

1.1 A PLD 2
1.2 Xilinx LCA 4
1.3 FPLD classification 6
1.4 FPLD design flow 7
1.5 FPGA cost advantage over MPGA 11

2.1 A Boolean Network 15
2.2 Decomposition 16
2.3 Look-up table (LUT) 17
2.4 Worst-case delay analysis 20

4.1 $\mathcal{M}_1$ implemented on $\mathcal{P}$ 29
4.2 $\mathcal{M}_2$ implemented on $\mathcal{P}$ 30
4.3 An RST 32
4.4 FPGA model 33
4.5 Internals of a qblock 34
4.6 Overview of RFR 36
4.7 Effect of integrating physical design and logic synthesis 42
4.8 Wire-length increase for constant qblocks 43
4.9 Routing congestion before and after running **RFR** 44

5.1 Symbolic representation of a 2-input LUT (TIL) 51
5.2 Example gate library with 4 gates 53
5.3 Tree covering for standard cells 54
5.4 Partitioning a DAG by a break at every multi-fanout node 55
5.5 3-input LUT as trees of 2-input LUTs 57
5.6 Library patterns for a 4-input LUT mapping 58
5.7 Input network → TINN 60
5.8 4-input LUT mapping 62
5.9 Reconverging fanout example 63
5.10 A 5-input LUT pattern 64
5.11 Example to illustrate fast tree-matching 66
5.12 Computing cost C_{13} 67
5.13 Computing cost C_{22}. 68

5.14 Computing cost C_{31} 69
5.15 Example to illustrate the fast matching drawback 71
5.16 **dpmap**'s area-delay trade-off solutions 73

7.1 XC4000 family routing resources 83
7.2 AT&T ORCA family routing resources 84

8.1 A general model of a sequential logic circuit. 91
8.2 Multi-level implementation of combinational logic block. 92
8.3 Signal propagation and identification of 'idle' resources 93
8.4 Levelize and hold 94
8.5 High level block diagram of *Dharma* 96
8.6 Detailed block diagram of *Dharma* 99
8.7 Clocking schemes 102
8.8 Circuits with $l > L$ can be implemented on multiple *Dharma*s 103
8.9 Dynamic logic module architecture 104
8.10 Dynamic interconnection block architecture 106
8.11 Block diagram of level generation circuitry. 108
8.12 Timing waveforms for multi-chip operation. 110
8.13 Timing diagram to illustrate *Dharma*'s operation 112

9.1 *Simple Dharma* 116
9.2 Schematic diagram for Example 1 116
9.3 DLM values for Example 1 117
9.4 Example 1. DIA: Primary inputs → level 1; DLMs : Level 2. 119
9.5 Example 1. DIA: Level 1 → Level 2; DLMs : Level 1. 120
9.6 Schematic diagram for Example 2 121
9.7 DLM values for Example 2 123
9.8 Example 2. DIA: Primary inputs → Level 1; DLMs : Level 4. 124
9.9 Example 2. DIA: Level 1 → Level 2; DLMs : Level 1. 125
9.10 Example 2. DIA: Level 2 → Level 3; DLMs: Level 2. 126
9.11 Example 2. DIA: Level 3 → Level 4; DLMs: Level 3. 127
9.12 Sequential Example 128
9.13 DLM values for the sequential example 130
9.14 Sequential Example. DIA: Primary inputs, state variables → Level 1; DLMs: Level 3. 131
9.15 Sequential Example. DIA: Level 1 → Level 2; DLMs: Level 1. 132
9.16 Sequential Example. DIA: Level 2 → Level 3; DLMs: Level 2. 133

10.1 An example of a complex LUT DLM 135
10.2 An example of a PLA based DLM 137
10.3 A heterogeneous DLM array 137
10.4 An example of L-crossbar which uses shift registers 139
10.5 Hard-wire connections from a DLM to adjacent DLMs. 142
10.6 Hard-wire connections reduce L-crossbar crosspoints. 143

10.7 A 15-to-4 binomial concentrator 144
10.8 DIA as a *Copier* 146
10.9 K-class DLMs reduce crosspoints 147
10.10A DIA with wire-AND capability 148
10.11An example to illustrate level expansion 150
10.12Modified *Dharma* with level expansion capability 152
10.13The *Local Level Generation* circuitry 153

11.1 Different levelization techniques 157
11.2 Temporal partitioning procedure 164
11.3 Move selection procedure 167
11.4 DLM distribution before and after temporal partitioning 169

12.1 Input to output signal path for EFP clocking scheme 183
12.2 Input to output signal path for S clocking scheme 185

13.1 Unbalanced and balanced implementation of function y. 193
13.2 Example of balanced decomposition. 194
13.3 Tiny *Dharma* with VSC. 196
13.4 An example DAG with repetitive structures. 198

List of Tables

4.1 Updating U_i and U_o, considering input nets 38
4.2 Updating U_i and U_o, considering output net 39
4.3 Results of running **RFR** for XC3000 45

5.1 3 level benchmark circuits 49
5.2 4 level benchmark circuits 49
5.3 Previous library sizes, for K-input LUT mapping 56
5.4 Number of K-input LUT patterns 57
5.5 **dpmap**'s performance-driven mapping results 72
5.6 **dpmap**'s minimum-depth mapping solutions 74
5.7 **dpmap**'s minimum-area mapping solutions 75

7.1 Dominant interconnect delay 86

11.1 DLM distribution statistics before and after temporal partitioning 170

12.1 Number of configuration bits 173
12.2 Number of pass transistors 174
12.3 Number of routing lines 174
12.4 Number of latches 174
12.5 *Dharma*'s area comparison with XC2064 176
12.6 *Dharma*'s area comparison with XC2018 177
12.7 *Dharma*'s area comparison with XC3090 178
12.8 Gate equivalent area comparison with XC3090 179
12.9 K-class DIA fares better 179
12.10 *Dharma*'s configuration bits comparison with XC4000 180
12.11 *Dharma*'s configuration bits comparison with AT&T ORCA 181
12.12 *Dharma*'s delay comparison with Xilinx 3000 series 188

Acknowledgements

I am grateful to my research advisor, Prof. Ernest. S. Kuh, for his guidance and constant support during my stay at Berkeley. His encouragement has been the driving force behind my graduate studies.

Prof. Robert. K. Brayton took on the responsibility of being my 'unofficial' research advisor, and guided my last year at Berkeley. He encouraged the *Dharma* idea and suggested that it be patented. I would like to thank him for many useful discussions, suggestions and help.

I thank Prof. Ilan Adler, of IEOR, for being on my qualifying examination and thesis committees. Thanks to Prof. Jan Rabaey for being on my qualifying examination committee and giving me useful suggestions.

Thanks to Dr. Dwight Hill for being my mentor at AT&T Bell Labs, and introducing me to FPGA research. Massoud Pedram taught me how to do research. I am indebted to him for his support and friendly guidance. Thanks to Kamal Chaudhary for many hours of stimulating discussions on a wide variety of topics.

I would like to thank Dr. Ulrich Lauther and Prof. Margaret Marek-Sadowska for helpful discussions. Thanks to Rajeev Murgai and Narendra Shenoy for useful feedback on my thesis proposals. Premal Buch, Charles Hough, Michael Jackson, Shen Lin, Minshine Shih, Arvind Srinivasan, Debbie Wang, Mei Xiao and Tianxiong Xue (David) made life in Kuh's group interesting and lively. Thanks to all of them. Thanks to Premal for reading my thesis. Special thanks to Tahani Sticpewich for helping me with travel grants, mailing, and most everything, and making life a lot easier. I would also like to thank George Carvalho, Vijay Maheshwari and Tan Wee-Chiew.

Special thanks to Sriram Krishnan, Amit Marathe, Rajeev Murgai, Jagesh Sanghavi and Narendra Shenoy for their friendship and company. Thanks to Deepak and Rekha, Arvind and Sandhya, and Kamal and Vijay for many enjoyable moments. My heartfelt gratitude to my local guardians (and cousins) Dr. Suresh and Rekha Nayak, and their family, for providing me with much needed emotional support and love, and giving me a home away from home. Thanks to my cousins Dr. Rajendra and Anita Bhat and their family.

My uncle, Dr. Umeshraya Pai, and aunt, Nalini, deserve special mention. Their love and sincere well-wishes are really appreciated. They carried their role of surrogate parents to the full. My parents, Balakrishna and Premaleela Bhat, and my brothers Prashanth and Srikanth, gave me monumental support, encouragement, and enthusiasm in my work.

My wife, Shashikala provided me encouragement and support, especially when I needed it most. I am forever indebted to all of them.

This research was supported in part by the SRC, under grant number 92-DC008 and 93-DC008. This support is gratefully acknowledged.

This work is a humble offering to my beloved parents

Sri Balakrishna and Smt. Premaleela Bhat

Chapter 1

Introduction and Overview

This thesis is about field programmable logic devices (FPLDs). As their name implies, these devices are integrated circuits that can be programmed (i.e., configured) by the user to implement digital logic circuits. First introduced in the mid-70's, they provided a means for rapid prototyping of simple (consisting of a few 100 gates) logic circuits. Over the years, these simple prototyping devices have evolved into sophisticated devices which can be programmed to implement logic circuits having several thousand gates.

Programmable logic technology was dominated in the past by devices based on a PLA-type architecture. The basic PLA architecture consists of a programmable AND block (popularly called AND plane) and a programmable OR block (OR plane). A given circuit to be implemented using such a device, is first represented in a two-level AND-OR form. The PLA AND-plane is then configured to implement the AND part, and the PLA OR-plane is configured to implement the OR part. Variations and evolutions of this basic architecture have resulted in several kinds of programmable devices. Some have a fixed OR plane instead of a programmable one; some use a NOR-NOR representation or NAND-NAND representation, some use more complex gates instead of a basic AND gate for the product term, similarly some use complex gates instead of the basic OR gate for the sum term and some provide features such as invertible outputs, registered outputs, configurable I/O, etc. All these devices, which *look* like a PLA, are popularly called PLDs. Figure 1.1 shows a logic diagram of a popular PLD.

With advances in IC technology, it has become feasible to produce larger and larger devices based on the PLA-style architecture. However, in terms of silicon usage and speed of circuit operation, it is not efficient to implement large logic circuits by means of

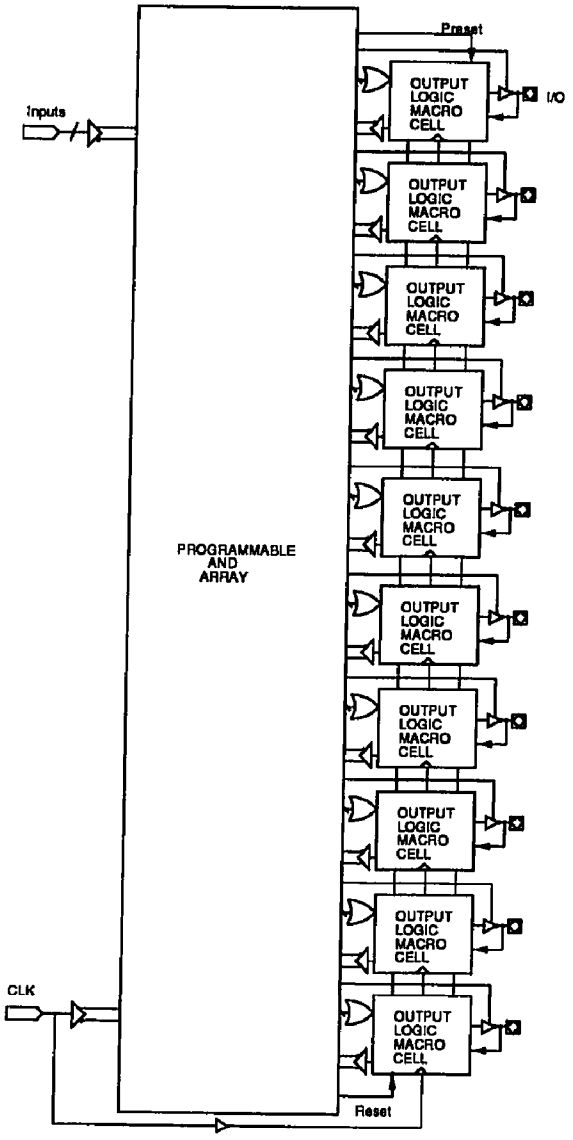


Figure 1.1: Logic diagram of the 22V10, a popularly used PLD

an AND-OR type of two-level implementation. In the case of large circuits, a multi-level implementation can be a better solution. Just as the foundry-based design styles evolved from the PLA into the standard cell and gate-array styles, so also the programmable logic technology evolved from PLDs into Field Programmable Gate Arrays (FPGAs).

FPGAs consist of a two-dimensional array of configurable logic cells, connected by means of a configurable interconnection network. The first of the current FPGA families, the logic cell array (LCA) was introduced by Xilinx, San Jose, CA, USA in 1985. In 1988 Actel introduced a configurable array architecture based on the structure of a channeled gate array. Today, the third generation of FPGAs is in the market. Figure 1.2 shows the Xilinx LCA.

FPLDs vary not only in architecture, but also in the programming technology. Early PLDs use bipolar technology, similar to PROMs, and are one-time programmable. These are manufactured with all connections intact, and programming consists of "blowing off" undesired connections. The configurable connections have a fuse link that is "blown off" in the programming process. With advance in technology, PLDs with erasable programming, like the Altera EPLD [Altera 85], are now available. These devices use EEPROM technology, and can be programmed several times. FPGAs also come in two flavors: the one-time programmable and the re-programmable kinds. Actel uses a new technology, called the anti-fuse, which is used to make a connection (as opposed to break a connection). The Actel devices are one-time programmable. Xilinx uses an SRAM technology to store the logic cell and interconnection configurations; the Xilinx devices can be programmed many times.

1.1 FPLD classification

There are several FPLD vendors, and many different kinds of devices in the market today. In Figure 1.3, an attempt has been made to classify the various FPLDs based on their underlying basic architecture. The chief purpose of Figure 1.3 is to give a feel for the different devices already existing, and also put in perspective, the new FPLD architecture being introduced in this thesis. It should be noted that Figure 1.3 is not intended to be complete, nor is it the only way to classify FPLDs. No standards for FPLDs have been established, and vendors have come up with ad-hoc nomenclatures that could be quite confusing. For example, a PLA-type architecture, with the output of the OR plane feeding

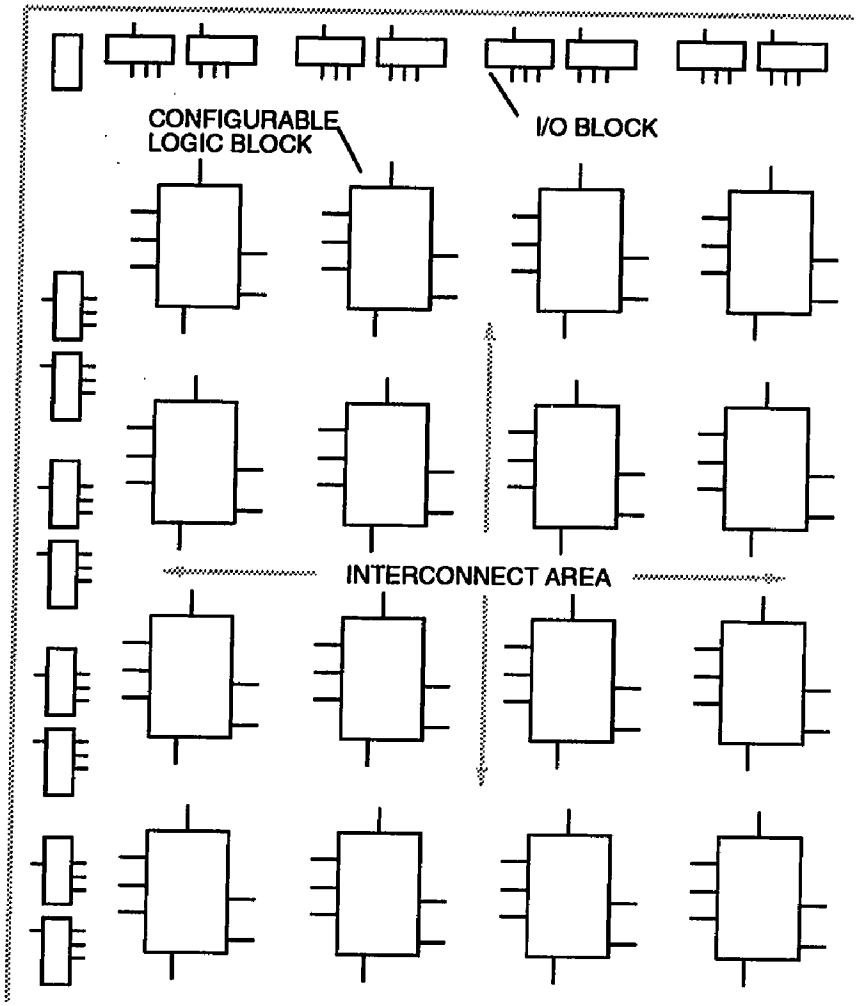


Figure 1.2: The Xilinx LCA

back into the AND plane can be used to implement multi-level logic circuits, and FPLDs with such an architecture are also called FPGAs by their vendors. However, for the purposes of this thesis, 'FPGA' is used to mean a device having a two-dimensional array architecture, consisting of configurable logic cells.

Figure 1.3 is based on Charles Small's [Small 91] high-density FPLD family tree. For this classification, only FPLDs with capacity to implement large logic circuits are considered. Prior to the new FPLD architecture of this thesis, the devices could broadly be classified into two: those that use the PLA-style architecture (classified as PLDs in the figure), and those that use the two-dimensional array (classified as FPGAs in the figure). PLDs can be further sub-divided based on the AND plane and OR plane architectures and FPGAs can be sub-divided based on the granularity of the logic cell. The *Dharma* architecture is a new programmable style, which performs a real-time re-configuration of the logic cells and interconnection patterns, so as to implement a multi-level circuit in a two-level manner. Detailed description of *Dharma* forms part II of this thesis. Since the *Dharma* style does not fall under either the FPGA or the PLD, a new branch at the root needs to be grown. It is possible that the *Dharma* architecture style will foster several devices varying in type of logic cell structure, interconnection techniques, etc.

1.2 CAD for FPLDs

The highly complex nature of current FPLDs necessitates the need for computer aided design (CAD) tools to design with, and program, such devices. The large number of possible interconnection strategies, and also the many different ways of distributing a given logic circuit among the configurable logic cells of FPLDs, makes it almost impossible to manage manually. Hence CAD tools become a necessary accessory to FPLDs.

Figure 1.4 is a flowchart to illustrate the various phases in the design of a circuit using FPLDs. In the figure, the logic optimization, mapping into FPLD modules, placement and routing, and the device programming steps are accomplished using CAD tools. The logic optimization and mapping steps are together referred to as logic synthesis. The design begins with a circuit specification, in terms of a register-transfer level description or a circuit schematic diagram. Using additional tools (not shown in Figure 1.4), one can also specify the circuit by means of a high level description (behavioral specification). The logic optimization tool takes as input the register-transfer level description. Logic

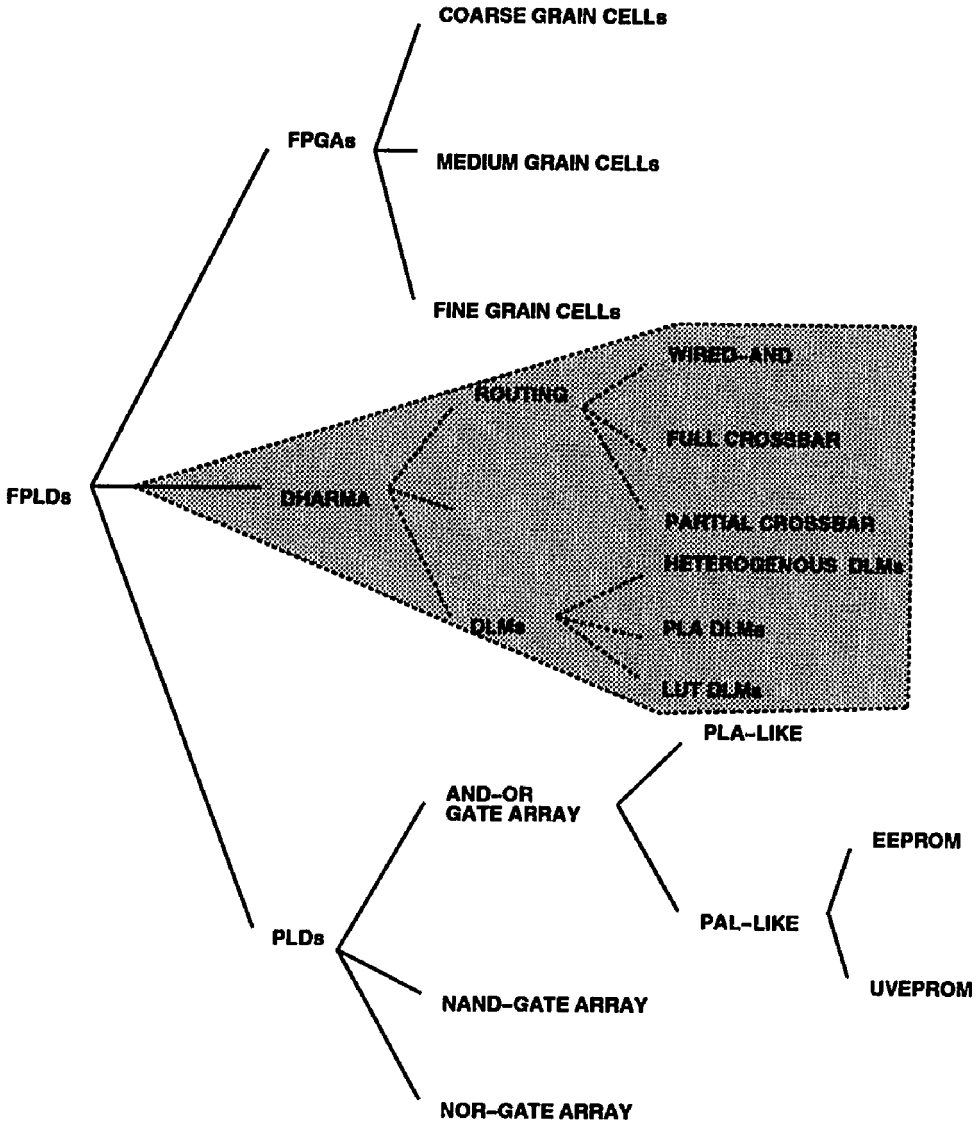


Figure 1.3: FPLD Classification

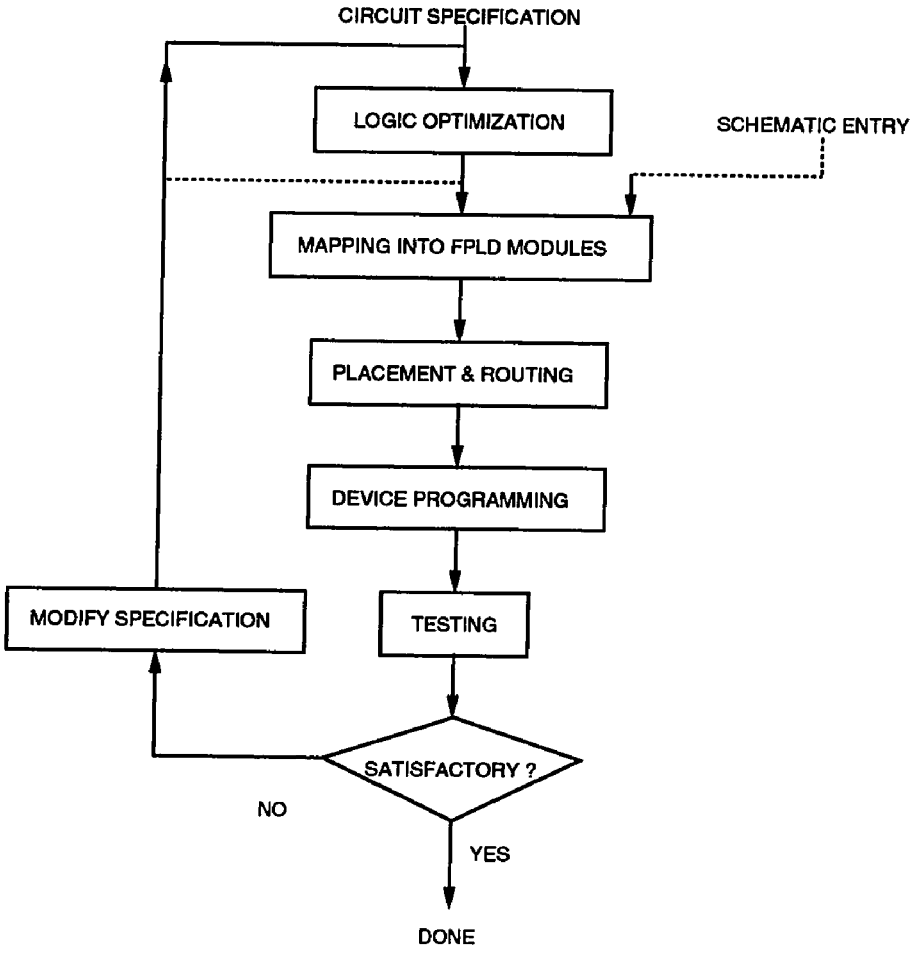


Figure 1.4: FPLD design flow

optimization is the process of manipulating the circuit structure and functionality of sub-circuits, while maintaining the specification, in a manner such that some parameters such as area of the overall circuit, and/or delay from inputs to outputs are optimized. A large body of work has been done in logic optimization for standard cell and gate array design styles [Brayton 90], and some FPLD optimization tools are based on these ideas. Some of these techniques, such as literal minimization (minimizing literals implies minimizing active cell area for standard cells and gate arrays) are not relevant to certain FPLDs such as LUT FPGAs, and in such cases, new optimization tools and techniques have emerged [Francis 92, Fujita 91, Murgai 90]. The next step in the design flow is mapping (also called packing or technology mapping). In this step, the architecture specific details are taken into consideration, and circuit structure is modified (maintaining the specification, of course) such that each gate or node in the circuit's network can be implemented by the logic modules of the FPLD on which the design is to be implemented. For PLDs, this step is usually simple and straight-forward. In the case of FPGAs, mapping is an area of active research [Bhat 92, Cong 92, Ercolani 91, Francis 91a, Francis 91, Karplus 91, Karplus 91a, Murgai 91a, Murgai 91, Murgai 90, Schlag 92, Trimberger 92, Woo 91]. Mapping is followed by placement and routing. In the placement step, the nodes in the mapped circuit are assigned positions on the FPLD (two-dimensional placement in the case of FPGAs), and the routing step figures out the path taken by the interconnections among the nodes. Following this, the actual programming of the FPLD is done (eg., loading 1's and 0's into the memory associated with the device, in the case of re-programmable FPLDs; or blowing fuses, anti-fuses in the case of one-time programmable FPLDs). The device now implements the circuit specification.

The steps of logic synthesis, placement and routing, and device programming are together called a design iteration. Since the circuit specification could be faulty (as humans usually tend to make errors), it may be necessary to go through several design iterations before a satisfactory circuit has been realized. The testing step in Figure 1.4 consists of plugging in the programmed device in the application it is intended for, and running several tests to check proper functionality [1]. If the test results are unsatisfactory, the specification is modified and another design iteration is performed. A typical design iteration could take about a day, on a personal computer such as the IBM 386 PC.

[1] In the case of fuse based devices, a "programming test" may also be necessary to make sure that the required links have been fused.

Figure 1.4 outlines only a typical design flow; modifications are possible. For example, in [Murgai 90], the authors recommend keeping the optimization and mapping together, as a single step. In [Bhat 92], mapping is combined with placement and routing. Vendor tools in the market do not perform logic optimization; the circuit specification (usually a schematic entry) is directly input to the mapping step (as shown by dashed lines in Figure 1.4), and the design iteration consists of mapping, placement & routing, and device programming. This is done because logic optimization tends to destroy the original circuit structure, which will make it almost impossible to pin-point design errors on the schematic.

1.3 Programmable logic system

The downside to the necessity of CAD tools in an FPLD based design is that the effective usage of the programmable device in a silicon-efficient and performance-optimized manner becomes the responsibility of the CAD tools. And unless CAD algorithms can use the special features of the FPLDs that they are assigned for designing with, the full potential of these devices cannot be effectively utilized. Because of the major role played by the CAD software in an FPLD based design, it is more appropriate to think of a programmable logic system (PLS), consisting of the FPLDs and their associated design tools.

Desirable features of a PLS are:

1. CAD tool features

   (a) *Fully automated*: In view of the rapid time-to-market requirement, manual intervention is not desirable.

   (b) *Fast*: In the absence of foundry fabrication time, and since programming the device takes the order of milliseconds, the time required to synthesize the design for the FPLD's architecture defines the design cycle time, and the CAD tools run-time is the real bottleneck in an FPLD design.

   (c) *Performance-driven*: The circuit to be implemented on the FPLD can have timing constraints, and the CAD tools must be able to honor these constraints.

2. Device features

(a) *Efficient silicon usage*: The programmable nature of the device requires routing and logic resources to be available in sufficient quantity so as to allow all possible circuits to be implementable on the programmable device. But excessive resources could result in silicon wastage. Less resources could result in unroutable circuits or excessively lengthy CAD tool run-times.

(b) *Low interconnect delays*: Programmable switches on the interconnection networks introduce significant delays, and can cause interconnection delays to exceed logic delays.

1.4 FPLD advantages

Along with increased complexity, there has also been a corresponding increase in the application areas for FPLDs. Application Specific Integrated Circuit (ASIC) users are now considering programmable logic devices as viable alternatives to foundry-based design styles such as gate arrays (now renamed as mask programmed gate arrays, MPGAs, to distinguish them from the FPGAs), sea-of-gates and standard cells. FPLDs offer two advantages for the ASIC user. The short design cycle of a PLS shortens the time-to-market, which is a critical factor in an ASIC's success. Also, since device costs are low, any design changes or design error fixes can be accommodated with negligible overhead (zero overhead for re-programmable devices). Gate arrays and sea-of-gate design styles require expensive mask fabrication, and changes and error corrections increase costs further. The non-recurring engineering (NRE) costs, which include mask fabrication and design change iteration costs, are absent in the case of FPLDs. Figure 1.5 shows the cost advantage of an FPGA [Xilinx 89] over foundry-based design styles. In addition to the ASIC market, re-programmable FPLDs have created new application areas such as hardware emulation [Quickturn 93], hardware acceleration [Hastie 90] and reconfigurable fast computers [Waugh 91].

1.5 Focus

This thesis focuses on the usage of FPLDs in the ASIC market, with the goal of extending their usage to applications which dictate timing performance. Complex circuits are now implemented on FPLDs instead of on gate arrays and sea-of-gates. However, in addition to being able to just *hold* the design within a programmable part, it is also necessary

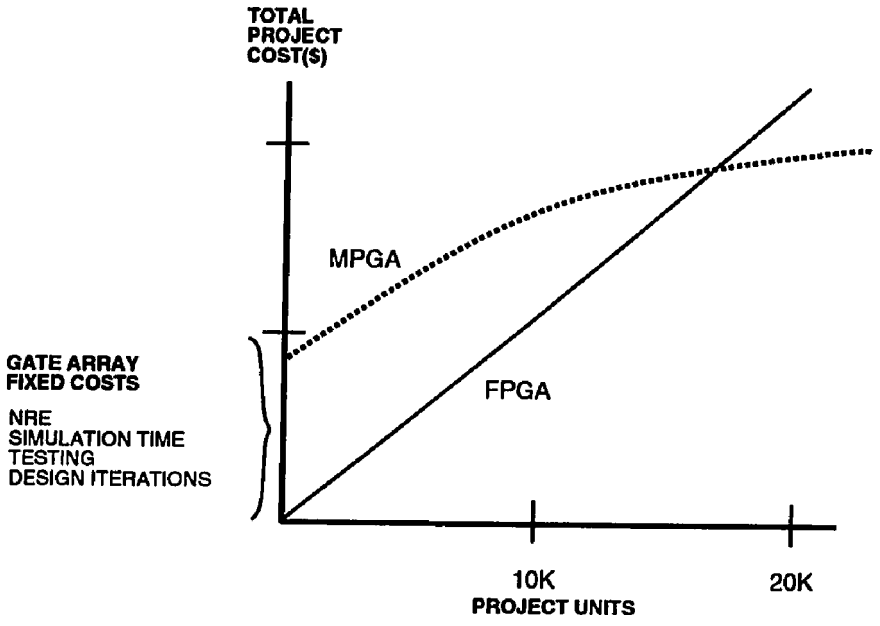


Figure 1.5: FPGA cost advantage over MPGA

to be able to perform the circuit functions fast. Only if the FPLD can meet the timing specifications of the logic circuit being implemented on the FPLD, can the FPLD be a viable alternative design style for ASIC users.

In this work, the emphasis is on the overall PLS. Both CAD and architecture aspects are considered. It may be possible to use existing FPLDs in high-performance applications by developing CAD tools that take timing performance into account when performing their design tasks. To explore this possibility, we choose a popular FPLD architecture, look-up table (LUT) FPGAs, and investigate the timing performance improvement by changing the structure of the mapped solution (of course, the logic function equality is maintained). In other words, we concentrate only on the technology mapping phase of logic synthesis, and propose a new performance-oriented mapping algorithm for FPGAs. For this part of the research, we restrict ourselves to combinational circuits only. This is justifiable since combinational circuits are easier to handle, and the ideas and algorithms can be easily extended to sequential circuits as well. The other possibility for overall PLS enhancement is to study architectural aspects. A new FPLD architecture, which is silicon efficient, performance-oriented and CAD friendly is proposed. As seen in Figure 1.3, this new architecture is different from existing FPLD devices and portends a new branch in the FPLD family tree.

We concentrate on single devices - i.e., we consider only those circuits that can be accommodated within a single programmable part. Circuits which cannot fit into a single part, will need to be partitioned across several parts, and this will require recourse to logic partitioning and physical partitioning techniques. Although these techniques have been extensively discussed in the literature [Donath 88, Yeh91] for standard cell and gate array based design styles, FPLD based partitioning techniques are still in their infancy, and merit further research.

1.6 Thesis overview

Definitions and glossary of more commonly used terms can be found in Chapter 2. The rest of this thesis is modeled after the PLS, in that it has two components. Part I has the software related work and Part II has the device architecture related work.

Part I deals with performance-oriented synthesis for LUT based FPGAs. By synthesis, we mean the mapping phase of Figure 1.4. The goal it to improve the performance of

a LUT FPGA, by modifying the mapped solution. Two aspects of the mapped solution are considered: ease of routing, and timing performance. The first sub-division of Part I deals with a routable mapping algorithm. Here, we combine the mapping step with the placement and routing step, using simulated-annealing. This approach yields mapping solutions that have lesser un-routed nets. The second sub-division of Part I presents a library-based performance-oriented mapping algorithm. This approach uses a concept of a 2-input LUT as a building block for the LUT library patterns and results in a very small library, several orders of magnitude smaller than previously documented LUT libraries [Francis 92]. This small library is used along with conventional library-based technology mapping techniques [Brayton 90] to give mapped solutions that have improved timing performance. The special nature of the library patterns is also exploited to yield faster pattern matching during the mapping process. The mapping algorithm also provides area-delay curve solutions, giving the user a family of solutions rather than a single *best-area* or *best-delay* solution.

Part II deals with the new *Dharma* architecture. We present the architecture, discuss CAD aspects for this new architecture and outline new research avenues. The central idea of *Dharma* is reconfiguration of the interconnections and the logic modules during run time (hence the architecture is *dynamic*). A multi-level circuit is implemented in a level by level manner, with inter-level connections accomplished using a crossbar interconnection network. The crossbar structure provides full routability, and the run-time reconfiguration allows reuse of the same crossbar to implement different net connections, making the architecture silicon efficient and practical. The net and logic delays are both predictable. This means that the synthesis step can make accurate estimate of the final (after device programming) timing behavior.

The final chapter concludes this thesis with a summary and future directions.

Chapter 2

Terminology

In this chapter, terms commonly used in the rest of the thesis are described. The chapter is meant to be used more as a reference, and hence the terms are arranged alphabetically. Terms and definitions specific to a chapter are defined in their respective chapters.

Boolean function

$B = \{0, 1\}$ is a Boolean space, and a variable that takes values 0 or 1 is a *binary variable.* A *Boolean function, F* maps an n-dimensional Boolean space to an m-dimensional Boolean space, i.e.,

$$F : \{0, 1\}^n \rightarrow \{0, 1\}^m$$

A Boolean function is also called a *logic function* or simply *function.*

The AND function is represented by a '$*$' or a space, the OR function is represented by a '+' and logical inversion is represented by a single quote after the variable, or an exclamation sign '!' before the variable.

Boolean Network

A *Boolean network* is a *directed acyclic graph (DAG)* used to represent a multi-level logic function. Each node i in the DAG is associated with a variable y_i and a representation f_i of a logic function. Figure 2.1 illustrates a Boolean network implementing the following multi-level logic function.

$$x = q + r'$$

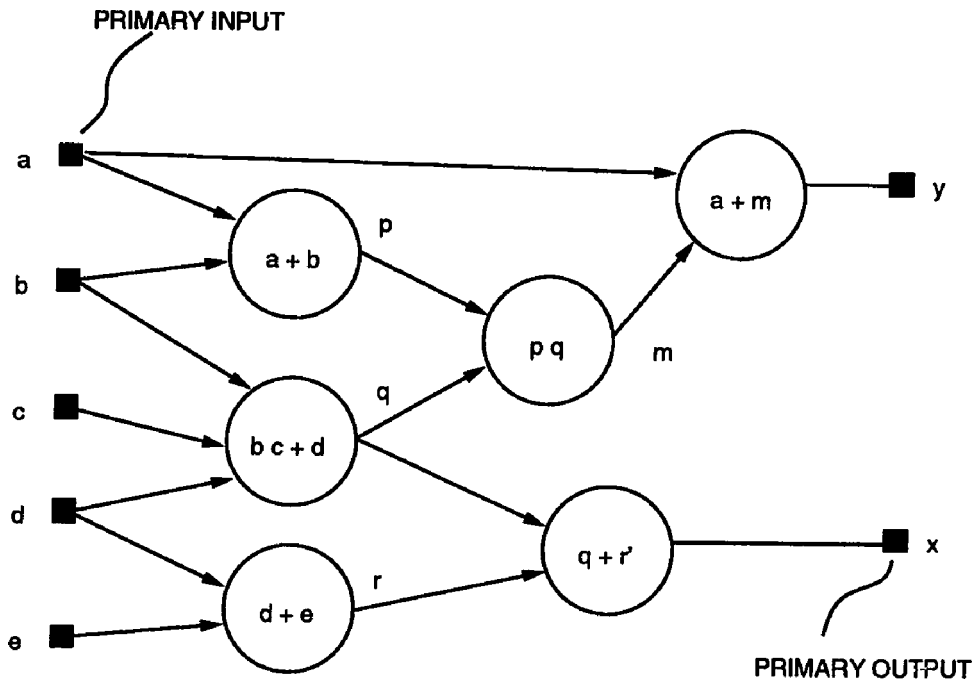


Figure 2.1: A Boolean Network is a DAG to represent a multi-level logic function.

$$
\begin{aligned}
y &= a + m \\
m &= p\,q \\
p &= a + b \\
q &= b\,c + d \\
r &= d + e
\end{aligned}
$$

In Figure 2.1, a, b, c, d, e are the PIs and x, y are the POs of the example multi-level logic function.

The *fanins* of a node n of a Boolean network is the set of nodes i whose output is directly connected to an input of n. In Figure 2.1, b, c, d are the fanins of node q. PIs do not have a fanin.

The *fanouts* of a node n is the set of nodes o that have an input directly connected to the output of n. m, x are the fanouts of q. POs do not have a fanout.

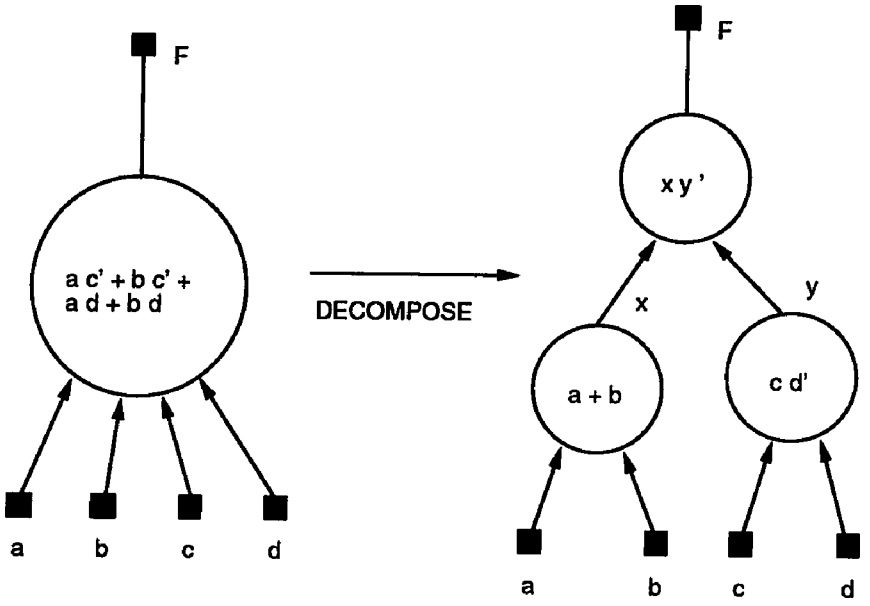


Figure 2.2: Decomposition is the process of re-expressing a single function as a collection of new functions.

DAG

DAG stands for Directed Acyclic Graph. A Boolean Network is a DAG.

Decomposition

Decomposition of a function is the process of re-expressing a single function as a collection of new functions. For example, re-expressing

$$F = ac' + bc' + ad + bd$$

as

$$\begin{aligned} F &= x\ y' \\ x &= a + b \\ y &= c\ d' \end{aligned}$$

is decomposition (see Figure 2.2).

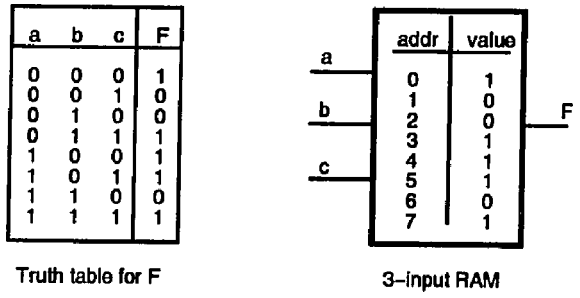


Figure 2.3: A look-up table (LUT) implementation of a 3-input Boolean function

Decomposition can be used to create a multi-level representation of a given logic function, specific to an application. For example, to represent the logic function as a DAG in which every node has $\leq m$ inputs, where m is a specified integer constant; or to represent the logic function as a DAG in which every node's function is either a NAND or INV; etc.

Logic module

A *logic module* is a piece of circuitry used to implement logic functions. In the case of LUT based FPLDs, like the Xilinx FPGAs, and the AT&T ORCA, logic modules are constituted from LUTs.

Look-up table (LUT)

A *look-up table (LUT)* is a random access memory (RAM) module, used to implement a logic function. Inputs to the function are presented as address value to the RAM. The function's value corresponding to a particular input combination is stored in the memory location addressed by that particular input combination. Thus, when presented with an input combination, the RAM "looks up" the value of the function it is implementing, and presents this value at the RAM output.

In Figure 2.3, a three input function F is specified by means of a truth table. On the right side of Figure 2.3, a 3-input RAM module, with values stored to implement F is shown.

An m-input, 1-output LUT has 2^m locations, each of which can be individually set to a 0 or 1, and therefore the LUT can be used to implement any function of m inputs.

An m-input, n-output LUT can implement any multiple-output Boolean function with $\leq n$ outputs and $\leq m$ inputs.

Multi-level logic

Multi-level logic refers to any multiple-output Boolean function represented by a set of interconnected functions. If F is the multiple-output Boolean function represented as a multi-level logic, the inputs to F are called *primary inputs (PIs)* and the outputs of F are called *primary outputs (POs)*. See Boolean Network for an example.

Packing

Packing is technology mapping in the context of FPGAs.

Performance

Generally, three metrics are used to measure the performance of an implementation: the silicon area used, the timing delay, and the power consumed. However, in this thesis, we use the term performance interchangeably with *timing* performance.

Placement

Given a netlist, i.e., a list of modules and the manner in which they are connected to each other, *placement* is the process of assigning a location to each and every module on a two-dimensional plane. Placement tries to place the modules in a manner such that after the subsequent routing step, the modules and interconnections use up minimal possible resources and/or the worst-case timing delay is minimized.

Primary input (PI)

See **Multi-level logic.**

Primary output(PO)

See **Multi-level logic.**

Routing

Given a netlist of placed modules, the *routing* process determines the path for the interconnections. Routing is usually performed in two steps: *global routing* and *detailed*

routing.

Technology mapping

Technology mapping or *mapping* is the process of representing a given Boolean network in terms of logic functions specific to a given technology. The software which performs this task is called the *mapper*. When the given technology is FPGA, this process is also called *packing*, and the corresponding software is called a *packer*.

The mapping process takes as input a Boolean network, called the *input* network and gives as output another Boolean network, called the *mapped* network. Each node in the mapped network can be implemented by the gates or logic modules of the technology for which the mapper is designed. For example, in the case of an m-input, 1-output LUT mapper, the mapped network has nodes with $\leq m$ inputs each.

Timing delay

The timing delay is a measure of the time required for signals to propagate through the physical implementation of a given circuit or Boolean network. A circuit's delay has two components: delay through the logic modules (gate delay), and delay through the interconnection (net delay).

We use a linear delay model to model the delay through a logic module. In this model, the delay from an input pin i of a logic module, to an output pin o is given by a linear equation,

$$I_{i,o} + R_{i,o}\Gamma$$

$I_{i,o}$ models the intrinsic delay through the module, from pin i to output o, Γ represents the capacitive loading at output o, and the coefficient $R_{i,o}$ represents the delay per unit capacitive load at o. The delay through the interconnection can also be incorporated into the above model. A net is approximated as a lumped capacitance, Γ_i. Γ_i is then added to the output load of the module which is the source of the net.

Consider the situation depicted in Figure 2.4. Let the signals *arrive* (i.e, settle to correct value) at pins a and b of logic module L at time instants t_a and t_b, respectively. L fans out to pins p and q of logic modules M and N, respectively. Let the arrival time of the output of L, x, at pins p and q be denoted by t_p and t_q, respectively. In the worst-case

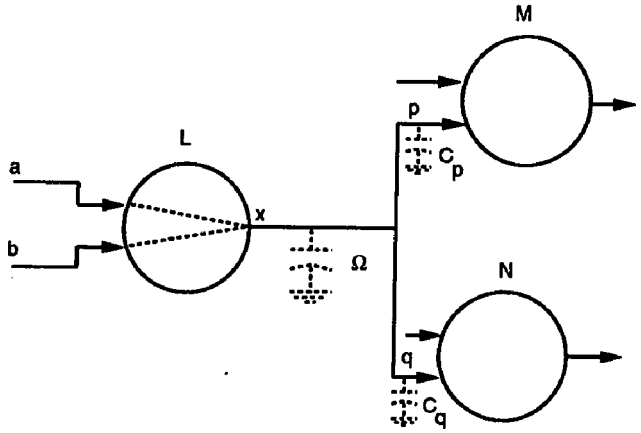


Figure 2.4: Example to illustrate worst-case delay analysis

delay analysis,

$$t_p = t_q = max\{(t_a + I_{a,x} + R_{a,x}\Gamma), (t_b + I_{b,x} + R_{b,x}\Gamma)\}$$

where,

$$\Gamma = \Omega + C_p + C_q$$

Ω represents the capacitive loading due to the interconnection, and C_p and C_q are the parasitic capacitances at the pins p and q, respectively.

The capacitive loading due to the interconnection can be modeled as being proportional to the number of sinks (input pins) on the net.

Part I

Performance-oriented Synthesis

Chapter 3

Overview

In this part of the thesis, the focus is on improving the CAD for existing FPLDs. As already mentioned, a PLS has a software component and a device component. The FPLD vendors have always introduced a device first, and then tried to develop the software to support the device. As a result, there are several FPLDs in the market, whose full potential has not been realized because their CAD tools have not yet matured.

The question we are attempting to answer is as follows. *Given an FPLD, how best can we design the CAD tools so as to get the maximum usage and performance out of this device?* This is a very general question, and because of the large number of different FPLDs, and different phases in the design flow (see Figure 1.4) it is almost impossible to have a single answer to this question. We therefore concentrate only on a single type of FPLD and only on certain phases of the design flow.

The most popular (in terms of number of units sold) FPLD is the LUT based FPGA (devices such as the Xilinx XC3000 family, XC4000 family and the AT&T ORCA family). We choose these kinds of devices as our "given FPLD". In the design flow, we concentrate on the mapping phase. Ideally, one could also look at the placement and routing phases. However, these require intimate knowledge of the device architecture, which is usually proprietary information. Also, placement and routing must be fine-tuned to specific architectures. Our purpose is to address a class of devices, hence it is not in the interest of this research to get bogged down by the idiosyncrasies of a specific device.

Two different aspects of the mapped solution are of interest. One is the timing behavior of the solution, and the other is the ease of routing.

3.1 Performance-oriented mapping

In applications where timing performance (henceforth called performance) is critical, the circuit specification is accompanied by timing constraints on the various output pins, in relation to the input pins. If the mapped solution is unable to meet these constraints, the device cannot be used for that particular application. For the purposes of our research, we impose the timing constraints in a slightly different manner. Given a circuit, our aim is to synthesize a mapped solution whose worst-case timing delay is minimal. In our experiments, all inputs are assumed to be present at the same time and the time delay at the latest arriving output signal is minimized. The reasons for this modification are as follows.

1. It is easier to benchmark different performance-improvement techniques, since the technique which yields the lowest worst-case delay is the better technique.

2. Our technique does not require timing constraints to be a part of the circuit specifications. Standard benchmark circuits do not have accompanying timing constraints.

3. Applications which supply timing specifications can still use the outcome of our research. If the mapped solution's timing delays are less than the specification's timing requirements, then the solution can be used for that particular application. If not, it means that the specifications have to be relaxed, since the solution provided is the best possible.

The delay through a circuit has two components: delay through the logic, and delay through the interconnection. With regard to LUT based FPGAs, prior efforts [Cong 92, Francis 91a, Murgai 91a] on performance-optimization have been concentrated on minimizing the delay through the logic. However, because of the limited amount of interconnection resources, LUT based FPGAs are characterized by dominant interconnect delays, which are difficult to estimate. Hence, mere logic delay minimization does not suffice. The delay of the final circuit (after placement & routing) is the relevant delay, and should be the objective being optimized.

A second consideration is that we are interested in addressing a *class* of devices, all based on LUTs. So, the mapper should be useful across the entire class of devices, with minor alterations to suit any particular architecture within this class.

A third consideration is that a family of mapped solutions is desirable. FPGA devices come in families, devices within a family differing in pin count and number of logic modules (larger devices are more costly). Circuit delay and circuit area (measured as the number of logic modules required to implement the circuit) bear an inverse relation: a circuit with better timing can only be obtained at the expense of more circuit area. However, if the mapper can provide a host of solutions, differing in area and delay, the user gets to choose a solution which is cost-effective and performance-optimized.

The above three requirements (accounting for net delay, using a single mapper across different architectures, and providing area-delay solutions) are not new to technology-mappers. Technology mappers used for standard-cell and gate array design styles have all the above features. Our idea is to extend these mappers to LUT FPGAs. However, these mappers use the concept of library-based mapping [Detjens 87, Keutzer 87]. Simple extension of the library-based approach to LUTs results in prohibitively large, impractical library of LUTs [Francis 92, Trimberger 92a]. We introduce the concept of a 2-input LUT as a building block (or primitive cell) for the library pattern, and demonstrate that this reduces library size by orders of magnitude, thereby making the library-based approach feasible and practical.

Using the new library patterns, and the old ideas of library-based mapping, our performance-oriented, library-based mapping algorithm can be set to optimize for depth (just to compare with previous work), delay, and to provide a family of solutions. Moreover, by simple alteration, the algorithm can be used for different LUT based architectures. However, our approach suffers the very bane of library-based mappers: sensitivity to initial decomposition. This disadvantage makes the area-minimization solutions inferior compared to other area-minimization based LUT mapping algorithms.

3.2 Routable mapping

Although it may seem that ease-of-routing has got nothing to do with performance-oriented synthesis, the mapping solution becomes useless if the placement and routing tools cannot complete the routing. In many instances, countless hours spent to manually route the circuit end up in vain because the mapping solution is truly impossible to route within the interconnection resource constraints of the device. Therefore, the first sub-division of this part of the thesis addresses this basic feasibility problem, and provides a technique to

alleviate the problem of unroutable circuits.

In our approach, we concentrate only on the objective of making the circuit routable, without regard to timing performance or area. Starting with a mapped solution to be placed and routed on a given FPGA device, we assume that the unused logic modules in the device are at our disposal, and we then re-distribute the logic, trading off logic block count for routability. Our algorithm integrates synthesis and physical design, much like [Pedram 91a, Pedram 91], but we use simulated annealing.

To make the integration of logic synthesis and physical design possible, we need to be able to treat the logic and routing resources in a unified manner. We model the FPGA architecture as an array of *qblocks* to be able to do this abstraction. This modeling forms the cornerstone of the routable mapping algorithm, which is called **RFR**, Re-synthesis For Routability.

3.3 LUTs and logic modules

In the next two chapters, we use the terms LUTs and logic modules interchangeably. In commercially available FPGA devices, logic modules are composed of interconnected LUTs. A logic module can have several outputs, and its constituent LUTs can share inputs. The manner in which the LUTs are interconnected, the number of inputs to the LUTs, and the number of outputs for the logic module varies from one family to another, and also from one vendor to another. Since our objective is to be able to address a class of devices, we restrict our mapping to consider only a single-output LUT. A network of single-output LUTs can be modified to fit into a specific architecture, using *merging* [Murgai 90] or library-based mapping (described in Chapter 5), and hence our restriction does not have any undue shortcomings. However, the library-based mapping approach of Chapter 5 cannot be extended to multi-output LUTs by any simple extension (therefore, *merging* will be required in such cases). The routable mapping technique does not depend on the LUT outputs or the logic module structure, and can actually be used for other types of FPGAs as well (non-LUT based).

Chapter 4

Routable Technology Mapping

This chapter concentrates on the routing feasibility of the mapped solution. As in the rest of this thesis, we have limited our attention to LUT based FPGAs. However, the ideas presented here are general enough that they can be extended to other types of FPGAs as well [1].

Our approach to the *routable mapper* is based on a philosophy of integrating physical design and logic synthesis. Our ideas are along the lines of [Pedram 91a, Pedram 91], where technology mapping and logic restructuring/decomposition are performed along with placement. Here, we go one step further, and include routing during the mapping process, using simulated annealing to tie all three (mapping, placement and routing) together.

We first motivate the need for taking routing into account. Terms used are then described. A key to make possible the integration of routing with mapping is the modeling of the FPGA architecture, and we introduce the concept of a *qblock*. Our routable mapper, **RFR** (Resynthesis For Routability) is then described. Experimental results, demonstrating the effectiveness of the approach, are presented next. The last section outlines shortcomings of our approach and discusses possible avenues to further this work.

4.1 Motivation

Routability is a primary concern with current FPGAs. This concern involves two factors: the routing needs to be 100% complete and correct within the given resources,

[1] Among currently available FPGAs, routability is a problem only for LUT based FPGAs, like the Xilinx XC3000. The 100% routability in other FPGAs is more because of greater availability of routing resources, and is not because of any routing driven mapping.

and it must not add so much delay that critical paths are delayed beyond the applications requirements. When CAD tools fail to provide a completely routed circuit, one has to manually intervene. Designers spend hours (and sometimes days) trying to complete the routing, by manually changing the routing and the distribution of logic among the FPGA logic modules. Manual intervention prolongs the design cycle time, and the FPGA 'rapid time-to-market' advantage is lost. Further, manual intervention does not always prove successful.

4.1.1 Cause of incomplete routing

There are only two reasons why a given circuit, to be realized on a given FPGA, results in a situation where some of the nets cannot be routed.

1. Existing CAD tools are not good enough.

2. Insufficient routing resources on the FPGA.

If the CAD tools are the best possible, and yet there are circuits which cannot be routed, then it is clear that the problem lies with the device itself, and its architecture needs to be altered to incorporate additional routing tracks and programmable connections. Our hypothesis is that the CAD tools are not good enough, and we proceed to improve them to enhance the possibility of obtaining fully routed circuits.

4.1.2 Problem with existing CAD tools

Let us examine the development of CAD tools for FPGAs. In Figure 1.4, we showed the design flow for FPGAs. This flow is very similar to the traditional design flow for standard cell and gate array design styles. In Figure 1.4, we see that mapping is followed by placement, and routing is done last. Thus, the synthesis and physical design tools have developed independent of each other. The assumption is that, whatever be the mapping solution, the placement and routing steps can provide correct solutions, be it in meeting timing specifications or in completely routing all nets.

This assumption is valid for standard cell and gate array designs. In such cases, an accurate estimate of the wire-length can be made [Kuh 90] using standard measures such as perimeter of bounding box, length of spanning tree, etc. A good router can then yield a routing solution that is close to these estimates. The problem of unrouted nets is

virtually absent, since the router has the flexibility of moving cell-rows apart to make room for additional routing tracks. Hence, the traditional design flow, of mapping followed by placement and routing, works well for these design styles.

The same is not true of LUT FPGAs, however. The router does not have the flexibility to move logic modules to make room for extra routing. It has to deal with a fixed amount of tracks and programmable connections. Even with the best possible placement and routing tools, it may still be impossible to complete the routing, because the routing was not accounted for, during the synthesis step.

As an example, consider a hypothetical FPGA, $\mathcal{P}$. Let each logic module in $\mathcal{P}$ be a 3-input, 1-output LUT. Consider the implementation of the following multiple-output function, F.

$$\begin{aligned} X &= a+b+c+d+e+f+g \\ Y &= f\,g \end{aligned}$$

Figure 4.1 shows one possible mapping, $\mathcal{M}_1$ of F, and $\mathcal{P}$ after mapping, placement and routing. Figure 4.2 shows a second mapping, $\mathcal{M}_2$ of F, and the corresponding $\mathcal{P}$ after placement and routing. By comparing Figure 4.2 and 4.1, we see that $\mathcal{M}_2$ uses lesser routing tracks compared to $\mathcal{M}_1$. Thus, this example shows that by changing the mapping solution we can change the routing characteristics.

4.1.3 Our strategy

We are motivated by the above example to improve the routability by changing the mapping solution. Previous techniques [Francis 91, Francis 90, Karplus 91a, Murgai 91, Murgai 90, Woo 91] for FPGA technology mapping have addressed the goal of minimizing the number of LUTs or configurable logic blocks (CLBs). In our approach, we diminished the relative weight of this constraint. Our motivation is to use the available resources (both routing and LUTs) for an FPGA of given size (where size is specified by number of LUTs) in an optimal manner. Current FPGA vendors have only a limited number of different sized FPGAs, and one can easily extend our algorithm to choose the smallest sized FPGA by stepping through the different sizes, arranged in ascending order, and picking the next higher size whenever the current choice is infeasible.

Our approach is based on performing mapping *along* with placement and routing. Each mapping step is followed by an incremental routing step. Thus, there is an *integration*

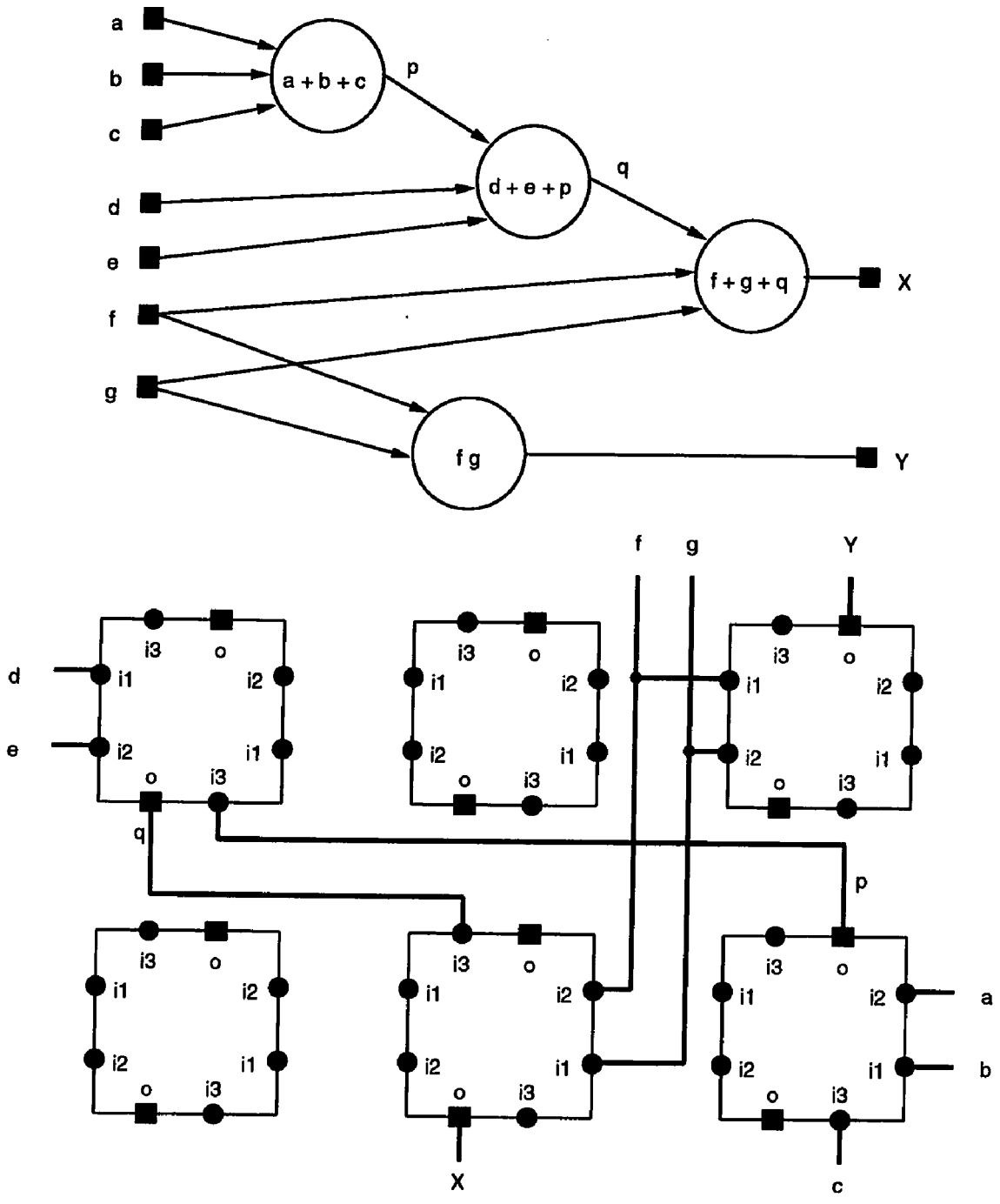


Figure 4.1: $\mathcal{M}_1$ implemented on $\mathcal{P}$

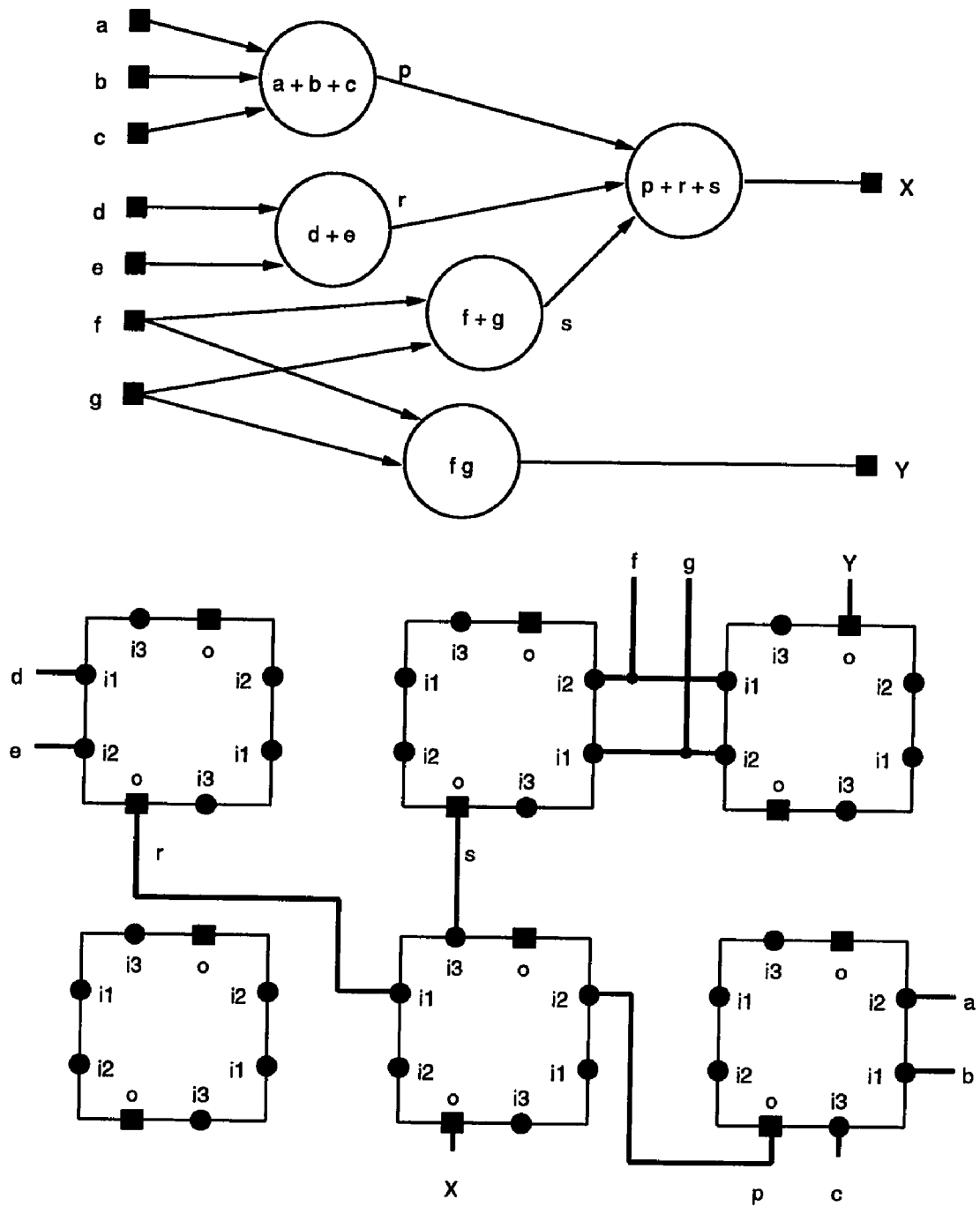


Figure 4.2: $\mathcal{M}_2$ implemented on $\mathcal{P}$

of physical design with logic synthesis, in our approach. This philosophy of integrating physical design and logic synthesis is relatively new. In [Pedram 91a, Pedram 91], the authors describe techniques to perform technology mapping, logic restructuring and decomposition along with placement. In [Abouzeid 90], the authors use an estimation of the routing factor to perform logic synthesis operations. In [Murgai 91], a placement based technique to perform timing optimized logic synthesis for FPGAs is used. Unlike our approach, however, none of these integrate routing with synthesis.

4.2 Terminology

A *qblock* is a single cell of the FPGA. The FPGA is constructed as a two-dimensional array of qblocks. Section 4.3 elaborates on the qblock.

Placking (PLAcement and paCKING) refers to the process of performing placement, routing and packing in tandem.

Iota of logic (IOL) is a small chunk of logic that is moved, or swapped amongst LUTs by the **RFR** algorithm. It can be a combinational function with a small number (1, 2 or 3) of inputs or a sequential element.

A packing is said to be *feasible* if the LUT can realize the logic assigned to it. Feasibility can be checked by comparing the corresponding number of inputs and outputs of the LUT and the logic function. e.g., it is feasible to assign a 3-input, 1-output function to a 4-input, 1-output LUT, but it is not feasible to assign a 5-input, 1-output function to the same LUT.

RST is an acronym for a *Rectilinear Steiner Tree*. An RST for a set A of points in the plane is a tree composed of vertical and horizontal lines which interconnect all members of A. Figure 4.3 shows a set of points $A = \{a, b, c, d, e\}$, connected by an RST. An *optimal* RST for A is one in which the lines have the shortest possible total length. Typically, there will be many optimal RSTs for A. The problem of finding an optimal RST is NP-complete [Garey 77], and many heuristic algorithms have been proposed [Hanan 66, Aho 77, Cohoon 90, Yang 72, Lee 76, Hwang 79, Lee 88, Ho 90].

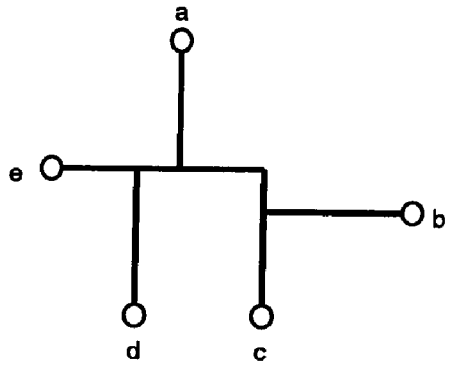


Figure 4.3: $A = \{a, b, c, d, e\}$ connected by an RST

4.3 FPGA Model

We first need to abstract the FPGA architecture in a manner such that placking is possible. Here, we introduce the concept of a *qblock*, which can model any FPGA architecture by merely changing its parameters.

The FPGA architecture that we have assumed for our algorithm has four types of components:

1. LUTs
2. Latches
3. Configurable interconnection points and
4. Routing tracks.

4.3.1 FPGA model

In our abstraction, we model the FPGA as an array of identical qblocks (see Figure 4.4). The qblocks are symmetric, and abut each other in the horizontal and vertical directions. At the abutment, the horizontal tracks abut in the horizontal direction, and the vertical tracks abut in the vertical direction, as seen in Figure 4.4. Not shown in the figure are programmable switches to split the tracks into separate segments. In addition

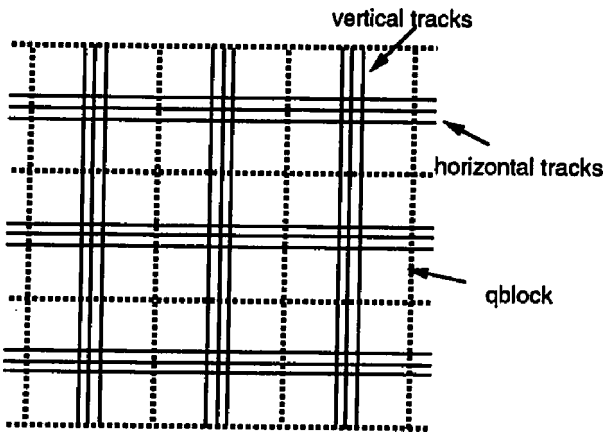


Figure 4.4: FPGA model. The FPGA is assumed to be an array of identical qblocks.

to these splittable tracks, there can also be continuous tracks (eg., long lines in the Xilinx architecture) and clock routing tracks.

4.3.2 qblock

Each qblock has the following resources.

1. Memory resources to implement LUTs
2. Latches for sequential logic
3. Routing resources
   (a) Horizontal tracks
   (b) Vertical tracks
   (c) Corners to connect horizontal and vertical tracks

Figure 4.5 shows a pictorial representation of this abstraction. The size of the memory, the number of inputs and outputs to the memory, the number of shared inputs, the number of latches and the quantity of routing resources are parameters of the qblock and can be set according to the commercial FPGA architecture being modeled. All qblocks are identical, in that they have the same number of resources.

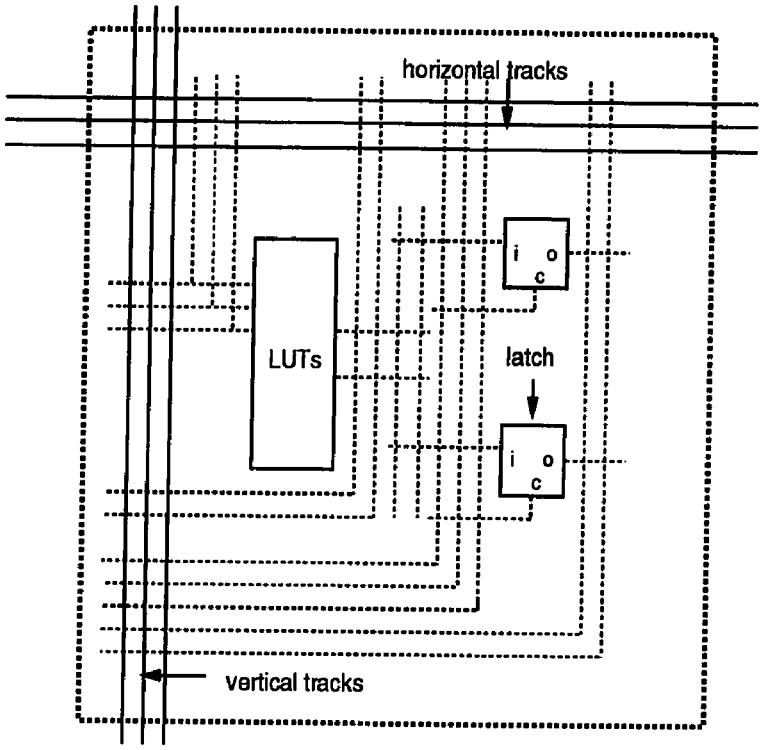


Figure 4.5: Internals of a qblock. The qblock has LUTs, latches and routing.

In Figure 4.5, the inputs and outputs of the LUTs and latches can be connected to any of the horizontal and vertical tracks, and this is illustrated by the extra routing lines (dotted). These extra lines are only for the purpose of illustration, the global router in our algorithm abstracts a qblock connection to lie directly on either the horizontal or vertical (or both) track(s). It is also assumed that the LUT outputs can be connected to latch inputs without going through the horizontal/vertical tracks.

4.3.3 Look-up table

The LUTs are configurable from the memory cells in a qblock. It is assumed that the memory cells are restructurable (by means of programmable switches), so that LUTs with varying number of inputs can be implemented. As an example, if there are 32 memory cells, then these can be structured as a 5-input, 1-output LUT, or a 4-input, 2-output LUT, or a 3-input, 4-output LUT, and so on. Additionally, two 4-input, 1-output LUTs (with no common inputs) and four 3-input, 1-output LUTs, and so on, are possible.

This model for the LUT is very general. We can restrict the number of inputs, number of outputs, number of shared inputs and the number of memory cells to model the LUTs of commercial LUT based FPGAs.

4.3.4 Routing resources

There are three kinds of routing resources: horizontal tracks, vertical tracks and corners. The number of such resources are fixed for each qblock, and all qblocks are assumed to have the same number of these resources. These are not supposed to represent the actual number of resources seen by the detailed router; they are for the purposes of global routing only. The algorithm is general enough to allow additional resources like connection points, long lines, etc.

4.4 Overview of our approach

We start from an initially optimized network, or from a mapped solution, eg., the output of Chortle-crf [Francis 91]. Each node in the input network is decomposed into IOLs. The IOLs are then re-organized in a manner such that the resulting network is easier to route, as compared to the input network. This re-organization is performed by swapping

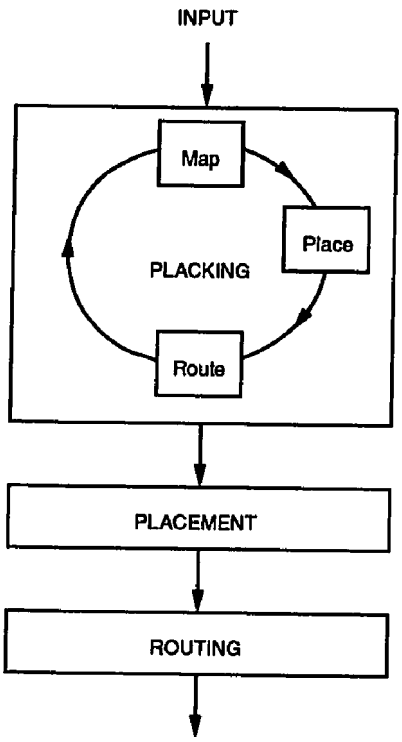


Figure 4.6: Overview of the RFR algorithm flow

IOLs, performing an incremental routing, and choosing the swap using a simulated annealing approach. After this re-organization, the new mapping solution is placed and routed using the vendor's commercial tools. Figure 4.6 shows the overall flow, and the placking step is explained in detail in the next section.

4.5 RFR algorithm

The algorithm for doing an integrated synthesis, placement and packing proceeds as follows.

1. Start from an initially optimized network [Murgai 90, Fujita 91].
2. Decompose the optimized network into a network of IOLs.

3. Pack the IOLs such that each cluster can be implemented using the latches and reconfigurable LUTs within a qblock. i.e., obtain a feasible packing.

4. Place the feasible packs on the FPGA plane, using some placement program, so that each pack is assigned to one of the qblocks in the FPGA (this step is done because we require an initial solution before we begin simulated annealing).

5. Perform an RST global routing. The global routing populates the horizontal, vertical and corner routing resources of each qblock.

6. Randomly pick an IOL, and evaluate the cost of swapping it with another IOL, or moving it to another qblock. By swapping, we mean that the IOLs swap both their positions and their qblocks.

7. Either choose or discard the swap, using a simulated annealing cost function.

8. Repeat the swapping process till no further congested channels remain (in case the design cannot be implemented within this FPGA, we will have to terminate after a sufficient number of iterations).

9. At termination, we will have a packed, placed and globally routed FPGA, which is ready for detailed routing.

4.5.1 Decomposition into IOLs

This is a crucial step, since the nodes in the decomposed network form the iota of logic that is swapped amongst clusters. The smaller the number of inputs, the greater is the possibility of performing feasible, cost reducing moves. We decompose into 2-input and 3-input nodes, allowing all possible functions, with a bias towards the more complex EXOR and EXNOR functions.

4.5.2 Global routing

An RST based global routing is performed using the separable minimum spanning tree to Steiner tree conversion algorithm presented in [Ho 89]. The qblock resources are populated based on the path of the route. For example, if a route starts at qblock located at (1,4) in the FPGA array, and runs horizontally to (3,4), and then vertically to (4,4),

I_k	Other fanouts	Consequence of removal	
		U_i	U_o
$\ni L$	none $\in L$	Decrement	No change
$\ni L$	at least one $\in L$	No change	No change
$\in L$	all $\in L$	No change	Increment
$\in L$	at least one $\ni L$	No change	No change

Table 4.1: Updating U_i and U_o, considering input nets.

the route is assumed to occupy one horizontal track in each of qblocks at (1,4), (2,4) and (3,4); one vertical track in qblocks at (3,4) and (4,4) and one corner in the qblock at (3,4). The global routing is performed sequentially on a net-by-net basis; each net is processed independent of the routing performed on the other nets before it.

4.5.3 Checking logic feasibility

Swapping of IOLs $I_1 \in A$ and $I_2 \in B$, where A, B are qblocks, is a 2-step process: (1) Remove I_1 from A and insert it into B. (2) Remove I_2 from B and insert it into A. At the end of the swap, A and B should be feasible, i.e, the LUTs and latches should satisfy the constraints of the target FPGA architecture.

Consider the case of combinational IOLs. For checking feasibility, we have to keep a running count of the number of "used-up" LUT inputs and outputs. When one (or both) of these exceeds the target architecture's constraints, we have an infeasible LUT. Since we perform the swap in 2-steps, we need to consider only the consequences of removing or inserting a single IOL from or into an LUT.

Let I_r be an m-input, 1-output IOL being removed from LUT L. Let U_i, U_o be the number of used-up inputs and outputs of L. For each input net of I_r, one of four possibilities can occur, as tabulated in Table 4.1. Three possibilities arise at the output net, as listed in Table 4.2. In Table 4.1, I_k is the source of the input net, and in the second column, 'Other fanouts' means fanouts (of I_k) other than I_r. 'Increment' means 'increment by 1' and 'decrement' means 'decrement by 1'.

From Table 4.1 and 4.2, we see that, as a consequence of removing I_r from L, U_i can increase by at most 1 and decrease by at most m and U_o can decrease by at most 1 and increase by at most m.

We can similarly list the possibilities for IOL insertion. These will be identical to

Fanouts	Consequence of removal	
	U_i	U_o
none $\in L$	No change	Decrement
all $\in L$	Increment	No change
some $\ni L$, some $\in L$	Increment	Decrement

Table 4.2: Updating U_i and U_o, considering output net.

Tables 4.1 and 4.2, except that the consequence of insertion will be the reverse of removal, i.e., in Tables 4.1 and 4.2, we replace 'increment' by 'decrement' and vice versa. In this case, U_i can increase by at most m and decrease by at most 1 and U_o can increase by at most 1 and decrease by at most m.

For sequential IOLs, feasibility check involves checking whether the number of sequential elements exceeds the qblock resources and also checking for compatibility of control signals.

The above method is suitable for simple LUT feasibility checking. Commercial FPGAs, however, have complex interconnection of LUTs which share certain inputs, with some outputs being connected to sequential elements, etc. In such cases, it is more suitable to have a *feasibility check function* specific to an architecture. The function takes as input a logic mapping and produces a yes/no output depending on whether or not the mapping is feasible. One such checking function will be needed for each commercial architecture, and the user is required to select the appropriate one.

4.5.4 Simulated annealing

The simulated annealing [Vecchi 83, Sechen 88] is straightforward. We start at a high temperature T, and evaluate $P = exp^{(-cost/T)}$, where cost is the cost of the swap. The swap is then chosen with probability P. The cost term consists of a weighted linear combination of two quantities: the wire-length and the congestion cost. The congestion cost is proportional to the difference between the congestion if the swap were implemented, and the present congestion. Congestion is quantified as the *number* of excess routing resources needed. At high temperatures, we can allow infeasible qblocks (i.e., the combinational function is allowed to have more than the allowed number of inputs/outputs, number of latches are allowed to exceed the logic cell quota, etc.). In such cases, the 'amount of infeasibility' is also combined into the cost function.

Let I, O, M, D be the number of LUT inputs, outputs, memory cells and latches respectively, as specified by the target architecture. Let U_i, U_o, U_m and U_l be the number of inputs, outputs, memory cells and latches used-up when a logic sub-network N is assigned to qblock Q. Then, the infeasibility in mapping N onto Q is quantified as $\text{MAX}(X_o, X_i, X_m, X_l)$, where $X_o = \text{MAX}(0, (U_o - O))$ is the excess number of LUT outputs, $X_i = \text{MAX}(0, (U_i - I))$ is the excess number of LUT inputs, $X_m = \text{MAX}(0, (U_m - M))$ is the excess number of LUT memory bits and $X_l = \text{MAX}(0, (U_l - D))$ is the excess number of latches.

Therefore, if $cost1 = \alpha W + \beta C$, and $cost2 = \gamma I$, where W, C and I are the wire-length, congestion and infeasibility costs, then $cost = cost1$ if the swaps do not cause infeasibilities, and $cost = cost1 + cost2$ if there are infeasibilities.

4.5.5 Re-synthesizing a packing solution

Instead of having a technology independent optimized network as input in step 1, we could start with an already packed FPGA and then make it routable. In this case, we decompose each pack into IOLs in step 2 and skip step 3. The rest of the algorithm is unaltered. Since we are swapping (or moving) IOLs around, the final logic network may have an altogether different structure, and the number of logic cells will usually change, when compared to the input network.

4.6 Experimental results

A version of the algorithm has been implemented in the SIS [Brayton 87, Sentovich 92] framework. The algorithm can run in two modes -

1. Perform placement, routing and mapping simultaneously.
2. Perform placement and mapping simultaneously (no routing at each swap). Here the cost of the swap is measured using a bounding box wire-length.

The second mode can be used as a pre-processing step before running mode 1 (the actual algorithm of Section 4.5), since it is much faster.

The second mode also allows us to illustrate the effect of integrating physical design and logic synthesis. Specifically, we carried out an experiment to demonstrate how

the algorithm makes a trade-off between the number of qblocks and total wire-length, as the size of the FPGA is increased. Figure 4.7 illustrates our observation. We implemented a benchmark circuit (`planet.kiss2` in the MCNC benchmark set) on FPGAs of different sizes, using simulated-annealing placement and packing. Since the input/output pins are restricted to the boundary of the chip, larger FPGAs will require larger total wire-length for implementing a given design. **RFR** reduces this increase in wire-length by increasing the qblock count. In Figure 4.7, the x-axis has the FPGA size in terms of number of qblocks, and the y-axis plots the qblock count and wire-length, normalized to their value corresponding to an FPGA of size 14×14.

If the qblock count were kept fixed, the total wire-length would increase with FPGA size, since the separation between the input/output pins would increase. This is illustrated in Figure 4.8. We perform the placement using simulated-annealing. In Figure 4.8, the x-axis has the FPGA size in terms of number of qblocks, and the y-axis plots the wire-length, normalized to the value corresponding to an FPGA of size 14×14.

In Figure 4.9 we show the results obtained by running the algorithm in mode 1. We start with an initial random solution, and with the routing congestion as shown, and then run the algorithm to yield a packing with less, or no, routing congestion. Figures 4.9(a) and (c) show the routing congestion before running **RFR**, and Figures 4.9(c) and (d) show the congestion after the algorithm was run. The results are for the MCNC benchmark circuit `ex1.kiss2` on a 10×10 FPGA. The congestion is shown by means of black rectangles within each qblock – the x-dimension of the black rectangle corresponds to the number of excess horizontal tracks and the y-dimension corresponds to the number of excess vertical tracks. In Figure 4.9(a) and (b), the FPGA has 8 horizontal and 8 vertical tracks per qblock. In this case, the algorithm is unable to remove all the congestion. We then increased the routing resources to 12 horizontal and 12 vertical tracks per qblock and Figure 4.9(c) and 4.9(d) show the congestion before and after running the algorithm, respectively. In this case, we get a congestion free solution. The total wire-length was 967 for 4.9(b) and 910 for 4.9(d).

The above experiments illustrate the effect of the algorithm for hypothetical FPGAs. We also interfaced **RFR** with Xilinx tools [Xilinx 89], to test the advantage of using the placking concept for a real, commercial FPGA. Our interface only handles combinational circuits.

We used the Xilinx XC3000 series chips as our target architecture. To test our algorithm, we needed examples with difficult routing characteristics. A few of the **Chortle-**

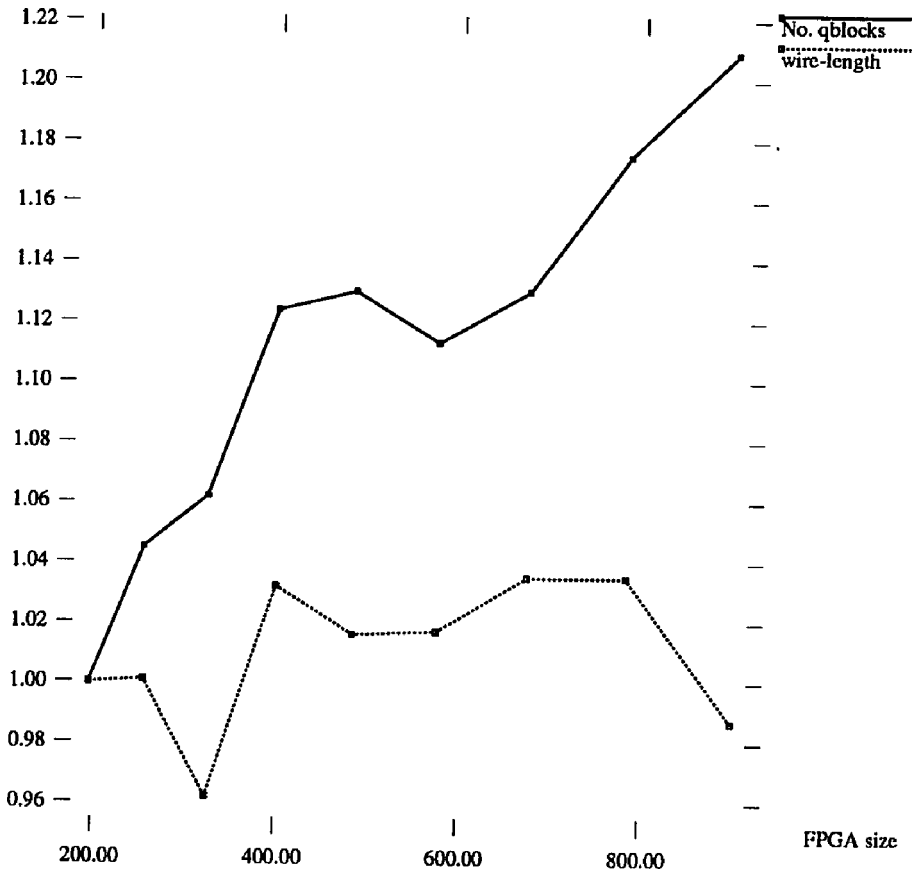


Figure 4.7: Effect of integrating physical design and logic synthesis. The algorithm trades-off qblock count for minimizing total wire-length. The x-axis has the FPGA size in terms of number of qblocks, and the y-axis plots the qblock count and wire-length, normalized to their value (of 177 and 2683 respectively) corresponding to an FPGA of size 14 x 14.

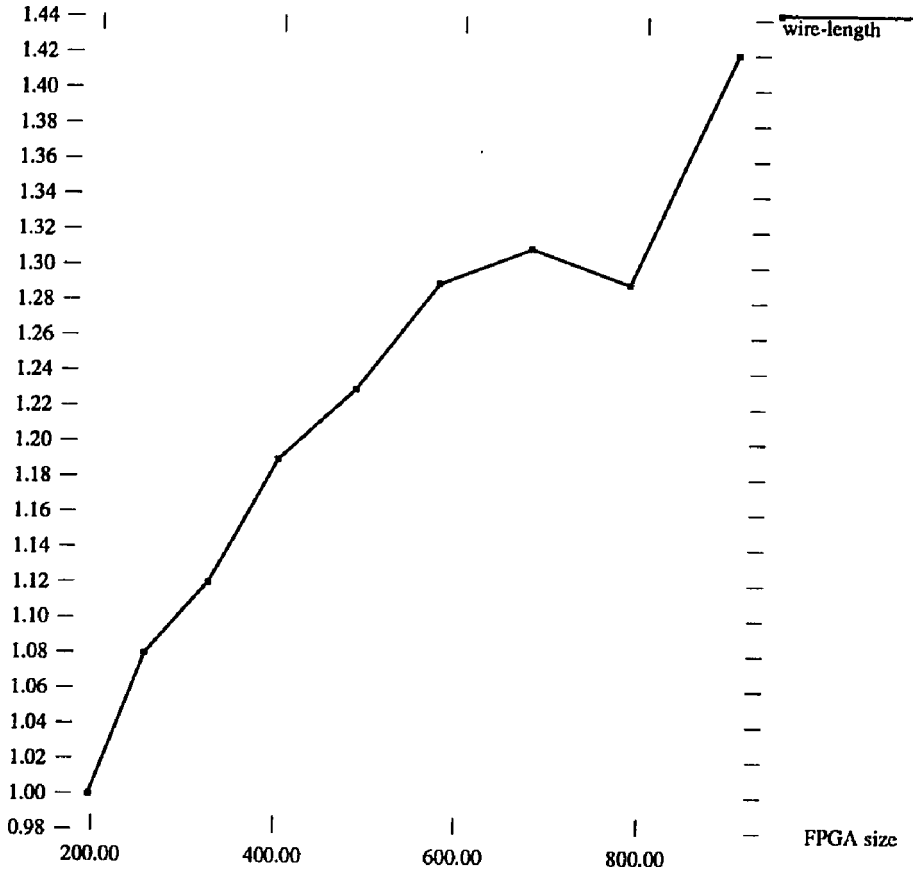


Figure 4.8: Illustrates how the wire-length would have increased with FPGA size, had the qblock count of the implementation been kept constant. The x-axis has the FPGA size in terms of number of qblocks, and the y-axis plots the wire-length, normalized to the value corresponding to an FPGA of size 14 x 14.

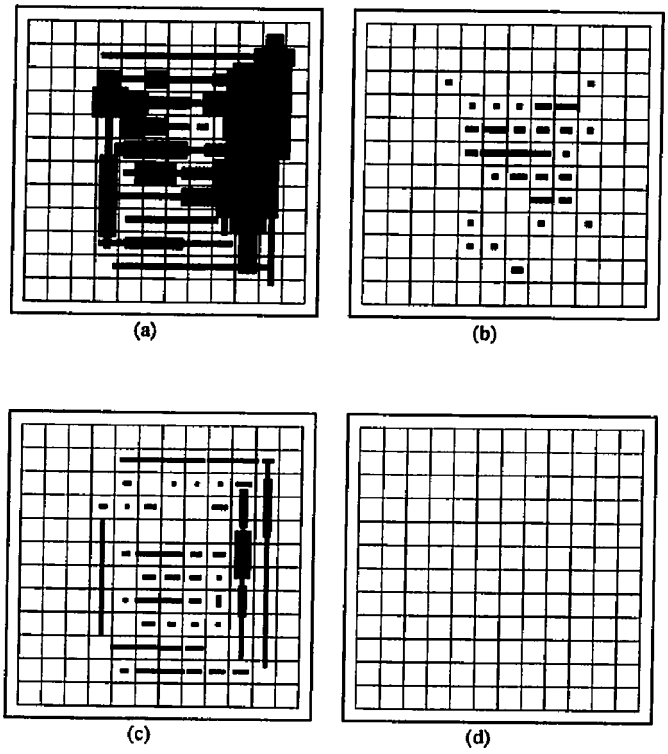


Figure 4.9: Routing congestion before [(a) and (c)] and after [(b) and (d)] running the algorithm in mode 1. (a) and (b) have 8 tracks, and (c) and (d) have 12 tracks. The total wire-length was 967 for (b) and 910 for (d).

mapped solutions [Francis 91] had such characteristics. i.e., when these examples were placed and routed on Xilinx 3000, using **apr** (Xilinx's place and route tool), some of the nets remained unrouted [2].

Our experimental procedure consisted of (1) reading in the **Chortle** mapped examples (2) performing placking (as described in 4.5.5) (3) and finally doing place and route using **apr**. We used $\alpha = 1.0$ and $\beta = 4.0$. Allowing infeasibilities did not help much. We used **Chortle**'s AND and OR nodes as our IOLs. Table 4.3 shows the results. The algorithm trades-off CLB count for routability. The CPU time shown is for the DEC 5000. The last column lists the Xilinx part used. duke2 and alu2 could also be fitted into 3030, but there was no improvement in routing, after **RFR**.

	Input circuit		After **RFR**			Xilinx Part
Example	CLBs	Nets	CLBs	Nets	Time (s)	(CLBs available)
duke2	92	15	108	8	3005	3042 (144)
alu2	93	7	101	2	3120	3042 (144)
alu4	152	51	171	17	4488	3064 (224)

Table 4.3: Results of running **RFR** for XC3000.

From the table, we see that there is a significant reduction in the number of unrouted nets (by a factor of 50%) or more, for all the 3 examples. The placking scheme is only partially effective, since the output of *RFR* still has unrouted nets. Since the IOLs are decomposed at the beginning, and the decomposition is kept fixed during the rest of the algorithm, it may be possible that the initial decomposition limits the number of different mapping solutions possible. Dynamically changing the decomposition, based on routing congestion information, during the course of the simulated annealing could make the algorithm more effective.

4.7 Conclusions

In this chapter, we presented heuristic techniques to alleviate the problem of obtaining routable FPGAs. These techniques are based on performing placement and packing in tandem, a process we call "placking." Global routing using RSTs is also done during placking, to measure routing congestion as the packing progresses. Simulated annealing

[2]The single output mapping solutions of **mis-pga** [Murgai 91] had very few unrouted nets for these examples.

has been used to bring together placement, routing and packing. Other techniques, like a greedy approach which chooses the best of all possible exchanges at each step, or one similar to Kernighan-Lin partitioning can be investigated.

We have not addressed timing issues directly (indirectly, by minimizing the wirelength, we decrease the delay in the route). An extension of this work would be to incorporate timing into the simulated annealing cost function. This would necessitate a timing analyzer to estimate the delay through an RST global route.

The final solution is dependent on the initial IOL decomposition. There are different ways to decompose the input Boolean network into IOLs, and we need to choose one that has favorable routing characteristics. Instead of keeping the decomposition fixed, one could dynamically vary it during the course of the annealing, using congestion and placement information.

Chapter 5

Performance-oriented Mapping

In this chapter, we present a mapping technique, for LUT FPGAs, which takes into account the delay in the nets during the mapping process. Hence the name performance-driven mapping. The algorithm is called **dpmap**, "dp" standing for dynamic programming (on which the algorithm is based). As outlined in Chapter 3, our approach extends the ideas of library-based mapping (LBM) used in standard cell and gate array designs. Since the LBM ideas form the core of **dpmap**, we first review the tree-covering approach used in standard cell mappers [Keutzer 87, Detjens 87], which use LBM. The important advantage of the tree-covering approach is that it can be used to optimize area, delay or their combination, just by changing the cost computation appropriately. Performance oriented technology mapping [Touati 90] and optimizing area under delay constraint [Chaudhary 92] are extensions of the library-based mapping approach. Why then, has the library-based approach not been used for LUT FPGA mapping? This question is answered in the following section. We then show that the problems associated with the LBM approach can be overcome by appropriate choice of a primitive cell, namely a 2-input LUT (TIL). We also show how the mapping process can be made faster by taking advantage of the special structure of the library patterns. This is followed by a description of our experimental procedure and results. Finally, we conclude with a list of shortcomings and possibilities to be investigated.

5.1 Motivation

Routing(net) delays are significant in LUT FPGAs and should be accounted for during the mapping phase. However, because of the inability of most LUT FPGA mappers

to handle net delays, current delay optimization is limited to depth optimization [Cong 92, Francis 91a], where the number of logic levels in the mapped network is minimized.

If a network has L levels of logic, the delay through the network t_D, using the delay analysis technique of Figure 2.4, is given by,

$$t_D = L \times t_{LM} + t_P + t_I,$$

where t_{LM} represents the delay through a logic module (this corresponds to $I_{i,o}$ in Figure 2.4, assuming identical intrinsic delay $\forall i$, and $\forall$ logic modules), t_P represents the delay through the input and output pads, and t_I represents the delay through the $(L+1)$ levels of interconnect. If t_I is ignored, then $t_D = L \times t_{LM} + t_P$, and minimizing L, minimizes t_D, since t_P is a constant quantity.

However, delay through the interconnection is significant, and at times dominates the logic delay. To demonstrate this point, we have tabulated the worst-case delays for some combinational benchmark circuits [1]. These delays are after place and route on the smallest Xilinx [Xilinx 89] XC3000 series chip that has enough number of CLBs to fit the design. Each logic module has 8ns delay (i.e., t_{LM} = 8ns), and input and output pads together have 33ns delay (i.e., t_P = 33ns). Two tables list our results, Table 5.1 and Table 5.2. All circuits in Table 5.1 have 3 levels of logic each, and all circuits in Table 5.2 have 4 levels of logic each.

Consider the 'Delay' column of Table 5.1. Although each circuit has 3 levels of logic, the delay varies by a factor of about 60%. The delay through the logic modules, t_{LM}, and the delay through the input and output pads is the same for each of these circuits, $8 \times 3 + 33 = 57ns$. But, the delay through the interconnect, t_I, is different and is the cause of this wide variation. We define a quantity, the routing delay per level, t_{RL}, given by

$$\begin{aligned} t_{RL} &= \frac{t_I}{(L+1)} \\ &= \frac{t_D - t_P - L \times t_{LM}}{(L+1)} \end{aligned}$$

For example, for the benchmark circuit **9symml**, t_D = 72.8ns, L = 3, t_P = 33ns, and t_{LM} = 8 ns, giving

$$t_{RL} = \frac{72.8 - 33 - 3 \times 8}{(3+1)}$$

[1] We have used MCNC [MCNC] benchmark circuits. In the tables, examples beginning with the letter 'c', except for clip and count, are combinational parts of sequential benchmark examples. For example, **cex2** is the combinational part of sequential benchmark **ex2**.

Circuit	Delay (in ns)	Routing delay per level (ns)
9sym	69.3	3.1
clion9	70.2	3.3
cs8	70.6	3.4
9symml	72.8	4.0
rd84	77.1	5.0
misex2	93.3	9.1
cex2	95.3	9.6
cdk16	102.8	11.5
b9	107.2	12.6

Table 5.1: Worst-case delays for some benchmark examples. All the circuits have 3 levels of logic each. Each circuit is placed and routed on the smallest Xilinx 3000 series chip that has enough number of CLBs to fit the design. The circuits have been arranged in ascending order of delays.

Circuit	Delay (in ns)	Routing delay per level (ns)
cex7	86.3	4.3
cex5	94.0	5.8
cex4	99.5	6.9
cs420	102.0	7.4
f51m	109.3	8.8
clip	120.5	11.1
count	133.1	13.6
apex7	168.6	20.7

Table 5.2: Worst-case delays for examples with 4 levels of logic each. Compared to Table 5.1, some circuits have lower delay than 3-level circuits.

$$= \frac{15.8}{4}$$
$$= 3.95 \text{ ns}$$

In the second column of Table 5.1 and 5.2, we tabulate the t_{RL} for each circuit. If t_{RL} were the same for circuits with same number of levels, then minimizing the number of levels would indeed minimize the total delay, t_D, and this is the implicit assumption made by depth minimizing algorithms. However, as seen in the second column, this is not the case; t_{RL} varies widely.

As a result of this variation in t_{RL}, we see that some circuits with 4 levels of logic (cex7, cex5, cex4 and cs420) have lesser delay than circuits with 3 levels of logic (cdk16, b9).

The conclusions from the above experiment are that the interconnection delay cannot be assumed to be a constant (hence, it cannot be ignored during the mapping), and that this delay could well be the dominating delay (eg., for apex7 in Table 5.2, $L \times t_{LM} + t_P$ = 65 ns, whereas t_I = 168.6 - 65 = 103.6 ns).

We proceed to account for the interconnect delay during the mapping phase, in a manner similar to the performance-driven technology mapping approach [Touati 90] used for standard cell design styles. This approach uses library-based mapping, which offers us three main advantages.

Firstly, we wish to have a single mapper that can be used across an entire class of devices based on LUTs. There are many such FPGAs in the market (eg., the Xilinx 2000, 3000, 3100 and 4000 series chips, and the AT&T ORCA chips) and more may be coming soon. Implementing a mapper from scratch for each architecture is not the desired approach. Handling several technologies in the case of standard-cell and gate array design styles has been achieved by using the library-based mapping approach. We would like to use a similar approach for LUT FPGAs.

Secondly, as already explained above, interconnect delay needs to be taken into account during mapping. The library-based approach provides a simple mechanism to account for net delay estimates during the mapping phase.

Thirdly, FPGA devices come in families, devices within a family differing in pin count and number of logic modules (larger devices cost more). The user must be given the flexibility to choose amongst these devices. Hence, the mapper should provide the user with the flexibility to choose from several different solutions, with different areas and

delays, instead of providing a single *best-area* [Francis 91, Murgai 91] or *best-delay* [Cong 92, Francis 91a, Murgai 91] solution. In other words, the mapper should provide a mechanism for area-delay trade-off. Again, the library-based mapping approach provides this facility.

5.2 Terminology

LBM stands for Library-Based Mapping. LBM is the process of implementing a given Boolean network (*input* network) using a given set of gates, called the *gate library.*

TIL is the acronym for a Two-Input LUT, having one output. Figure 5.1 shows the symbolic representation of a TIL.



Figure 5.1: Symbolic representation of a 2-input LUT (TIL).

TINN stands for a Two-Input Nodes Network. It is a network in which every internal node has exactly 2 inputs. There is no restriction on the function being implemented by a node.

A *primitive cell* or *base cell* is a simple function or structure, from which every other function or structure can be constructed. For example, in the case of standard cell mapping, the input network, and the library gates are represented as a network of INVs and 2-input NANDs. The INVs and 2-input NANDs are primitive cells.

A library *pattern* is a representation of a library gate in terms of primitive cells.

The *subject graph* or *subject network* is the representation of the input network in terms of primitive cells.

5.3 Review of library-based mapping

In the case of standard cells and gate arrays, technology mapping is the process of implementing a given Boolean network (*input* network) using gates from a given library.

A successful technique for solving this is based on an approximation of the DAG covering problem, first proposed by [Keutzer 87]. The input network is decomposed into a forest of trees, and each tree is optimally covered using patterns representing the library gates. In this section, we review this basic tree-covering algorithm. The **dpmap** algorithm uses this very algorithm, and hence this background is essential to understand the new approach.

5.3.1 Tree covering algorithm

The first step is to represent the input network in terms of *primitive* cells (eg., 2-input NANDs and INVs). This network, consisting only of 2-input NANDs and INVs, but logically equivalent to the input network, is called the *subject graph.* Each library gate is represented as a tree of primitive cells (if there are many such trees, each tree is stored). The gate representations will be called library *patterns.* Associated with each library pattern is a *cost.* The subject graph is partitioned into a forest of trees, and each tree is then optimally covered by the library patterns [Detjens 87]. The optimal covering is based on a dynamic programming algorithm.

In Figure 5.2, we show library patterns for an example library consisting of 4 gates, viz., 2-input NAND (nand2), an INV (inv), 3-input NAND (nand3) and a 3-input AND-OR-INVERT (aoi). These gates are represented in terms of the primitive cells, 2-input NAND and INV. For ease of explanation, we assume every gate has unit area cost.

The dynamic programming based covering algorithm proceeds from the leaves to the root (i.e., depth-first traversal). At each node, all the library patterns that match at this node are enumerated, and the cost of each match is evaluated. The matching pattern with the least cost is stored at the node. After processing the root node, in this manner, a simple back-tracing will yield the mapped solution.

Consider the mapping of Boolean function

$$Y = a'c' + b'c' + d + efg + h$$

using the example library of Figure 5.2. In Figure 5.3, on the left half, we show the subject network corresponding to Y, and on the right half, we show the mapped network, where each shaded region corresponds to a gate in the example library. The numbers on the left correspond to the best cost stored, for a minimum-area cost mapping.

The cost computation is based on the metric being optimized. For example, for a

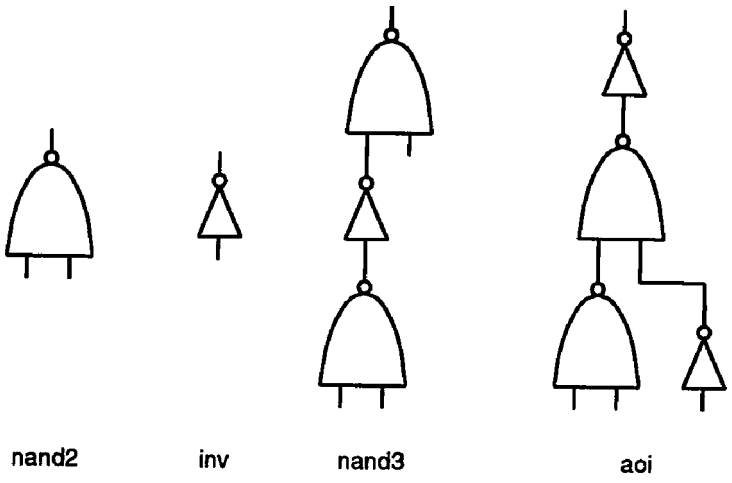


Figure 5.2: Example gate library with 4 gates. The library patterns are trees of INV and 2-input NANDs.

minimum-area solution, the cost of the match, m, at a node, n, is given by

$$cost(m) = area(m) + \sum_{v_i \in inputs(m)} cost(v_i).$$

Primary inputs have zero cost.

For a minimum-depth solution,

$$cost(m) = 1 + \max_{v_i \in inputs(m)} (cost(v_i)).$$

Primary inputs are at zero depth.

Using the delay model of Chapter 2, a minimum-delay solution can be obtained by computing the cost as

$$cost(m) = \gamma \times \#fanouts(n) + \max_{v_i \in inputs(m)} (cost(v_i) + delay_{v_i}(m)),$$

where γ is an estimate of the delay per fanout. $delay_{v_i}(m)$ is the propagation delay from input v_i to the output, for gate m, and $\#fanouts(n)$ is the number of fanouts of node n.

A combination of area and delay costs can be computed by using a linear combination, eg.,

$$cost(m) = w \times area\_cost(m) + (1 - w) \times delay\_cost(m)$$

where $0 \leq w \leq 1$.

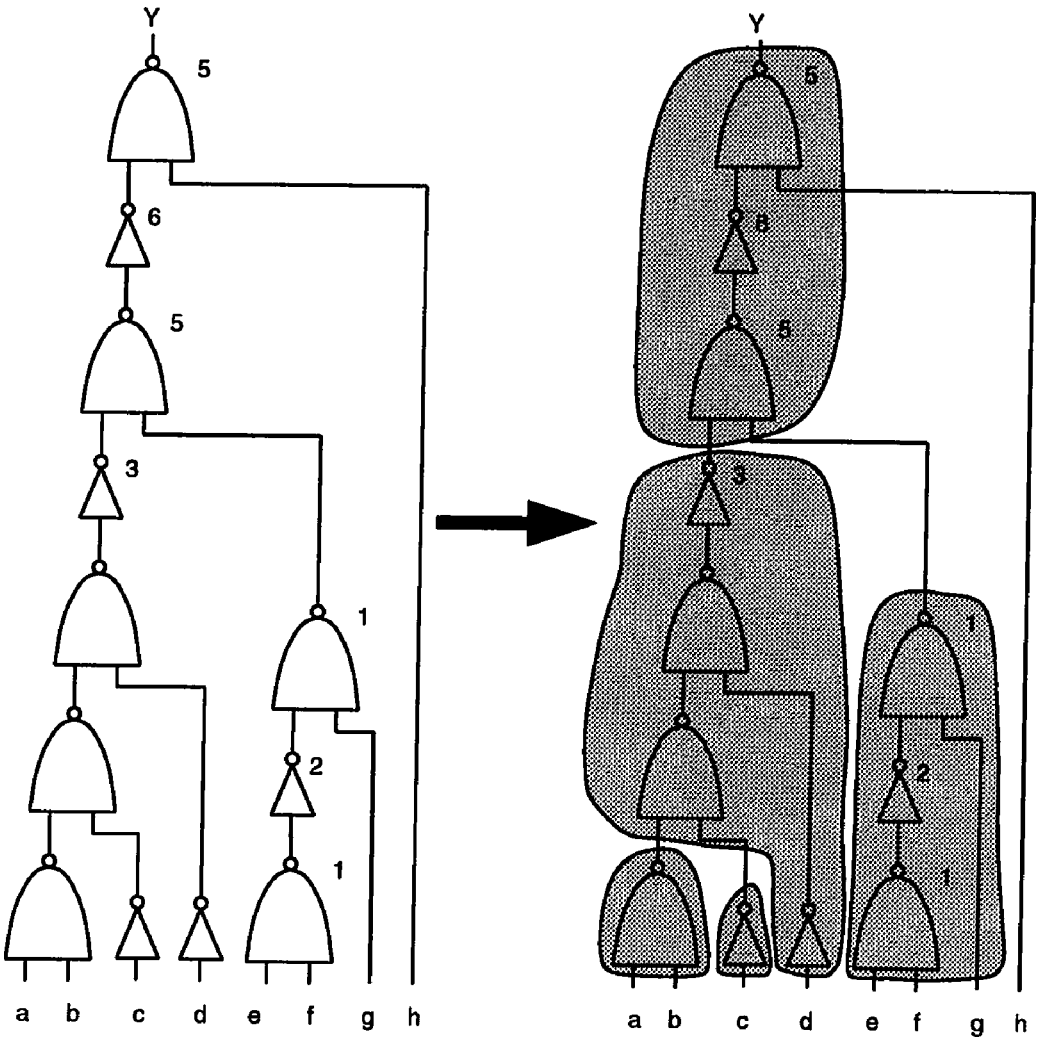


Figure 5.3: Tree covering for standard cells.

5.3.2 Partitioning DAGs

There are many ways in which the input network can be partitioned into a forest of trees. One approach is to partition at every node that has multiple fanouts (eg., see Figure 5.4). This could result in a lot of small trees, and the approach does not allow logic duplication. Another approach, single-cone partitioning, uses only primary outputs as roots of trees [Detjens 87]. This allows logic duplication (where the trees overlap). Usually, this approach yields better results for timing optimization.

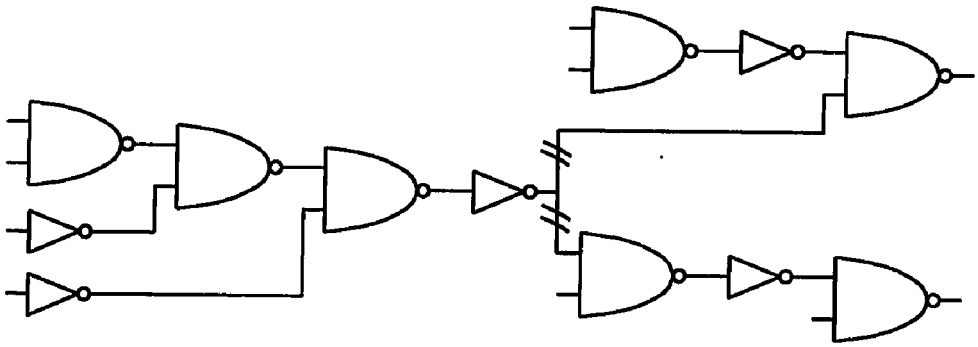


Figure 5.4: Partitioning a DAG by a break at every multi-fanout node

5.4 Why has this not been done before?

There have been attempts to extend the LBM approach to LUT mapping [Francis 92]. However, these did not meet with much success, since it was thought that the number of library patterns required to represent the LUT would be impractically large.

To quote from [Francis 92]: *The major obstacle to applying library- based technology mapping to LUT circuits is the large number of different functions that a K-input LUT can implement. The function implemented by a K-input LUT is determined by the values stored in its* 2^K *memory bits. Since each bit can independently be either 0 or 1, there are* 2^{2^K} *different Boolean functions of K variables. For values of K greater than 3 the library required to represent a K-input LUT becomes impractically large.*

A 4-input LUT can implement all the 65,536 functions of 4 variables. A 5-input LUT can implement all the 4,294,967,296 functions of 5 variables. The number of patterns

to be stored in the library can be reduced by noting that some patterns are equivalent after a permutation of inputs. Further reduction is possible by allowing functions differing by input or output inversions to share the same representation. In spite of this reduction, the number of functions is still impractically large. For the reader's convenience, the table of pattern count from [Francis 92] has been reproduced here as Table 5.3.

K	without permutations and inversions	with permutations	with permutations and inversions
2	16	12	4
3	256	80	14
4	65536	3984	232

Table 5.3: Previous library sizes, for K-input LUT mapping

5.5 TIL based LUT library

Previous attempts to use LBM for LUT mapping implicitly assumed that the library patterns would have to be represented in terms of 2-input NAND and INV primitive cells. However, the tree covering algorithm does not dictate the choice of primitive cells.

By choosing the primitive cells appropriately, the number of library patterns can be significantly reduced. This section shows how to use a 2-input LUT (TIL) as a primitive cell, and the advantages thereof.

5.5.1 TILs as primitive cells

A TIL can implement all functions of 2 variables. Hence it can *cover* any node with 2 inputs. Figure 5.1 introduces the symbol for a TIL.

A 3-input LUT can be decomposed into a tree of TILs. There are only 2 such trees: one with the root node having a left child as leaf and right child as a node with 2 children leaves; and the other with the root node having a left child as a node with 2 children leaves and the right child as leaf. Figure 5.5 shows these decompositions.

A 4-input LUT can similarly be decomposed into a tree of TILs. There are only 5 such trees. Figure 5.6 enumerates all the tree patterns that need to be stored in the mapping library, for a 4-input LUT based FPGA mapping. This library includes the 3-input and 2-input LUT patterns, since functions with 2 or 3 inputs can also be implemented in a

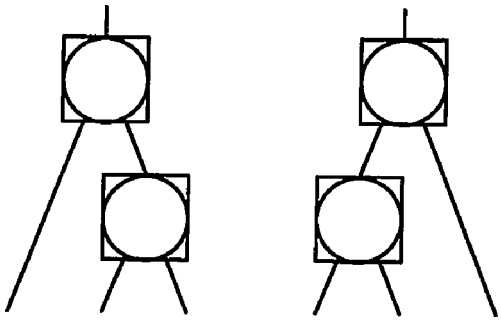


Figure 5.5: 3-input LUT as trees of 2-input LUTs

K	K-input LUT Patterns	Total Patterns
2	1	1
3	2	3
4	5	8
5	14	22
6	42	64
7	132	196
8	429	625
9	1430	2055
10	4862	6917

Table 5.4: Number of patterns required to be stored for various values of K, where K is the number of LUT inputs.

4-input LUT. There are 8 patterns in all (For a K-input LUT mapping, patterns for all LUTs with $\leq K$ inputs must be stored).

5.5.2 Small library

Table 5.4 tabulates the number of patterns per K-input LUT, and the total number of patterns required to be stored in the library, for values of $K \leq 10$. The number of patterns required for a K-input LUT is given by

$$N(K) = \sum_{i=1}^{K-1} N(i)N(K-i),$$

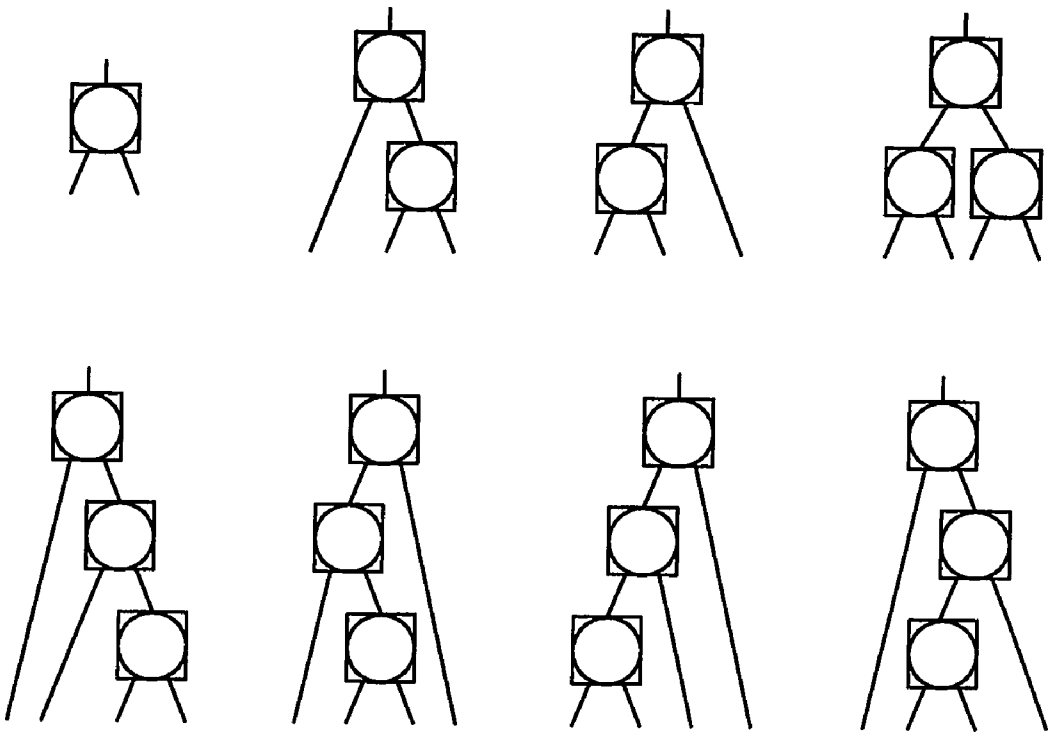


Figure 5.6: Library patterns for a 4-input LUT mapping

with $N(1) = N(2) = 1$. A K-input LUT library will have $S(K)$ patterns, where

$$S(K) = \sum_{i=2}^{K} N(i).$$

Compared to Table 5.3, we note that this library is significantly smaller.

5.6 dpmap Algorithm

To use the concept of the TIL as primitive cell, the subject graph should be such that each node in the subject graph can be covered by any TIL. In other words, every node in the subject graph should have exactly two inputs. The node can have any function, the only restriction is to limit the number of inputs to 2. Such a graph is called a TINN (for Two-Input Node Network). Once the subject graph has been converted into a TINN, the tree covering algorithm can be used to cover the TINN by the LUT patterns (of the previous section). The complete mapping procedure therefore has the following main steps.

1. Convert the input network into a TINN.
2. Represent the LUT in the manner shown in the previous section.
3. Use the tree-covering approach to cover the TINN with the LUT patterns.

5.6.1 Input network → TINN

As a simple extension of the standard cell based technique, the input network can be converted into a TINN in the following manner (see Figure 5.7).

1. Decompose the input network into 2-input NANDs and INVs, using the same technique as for standard cell technology mappers.
2. Absorb each INV into its fanout or fanin gate, whichever is convenient (eg., if an INV fans out to multiple nodes, absorb it into its fanin).

An alternate technique would be to have a complete 2-input library, as used by conventional standard cell mappers. The input network is first mapped into the gates of this library, using a standard cell technology mapper.

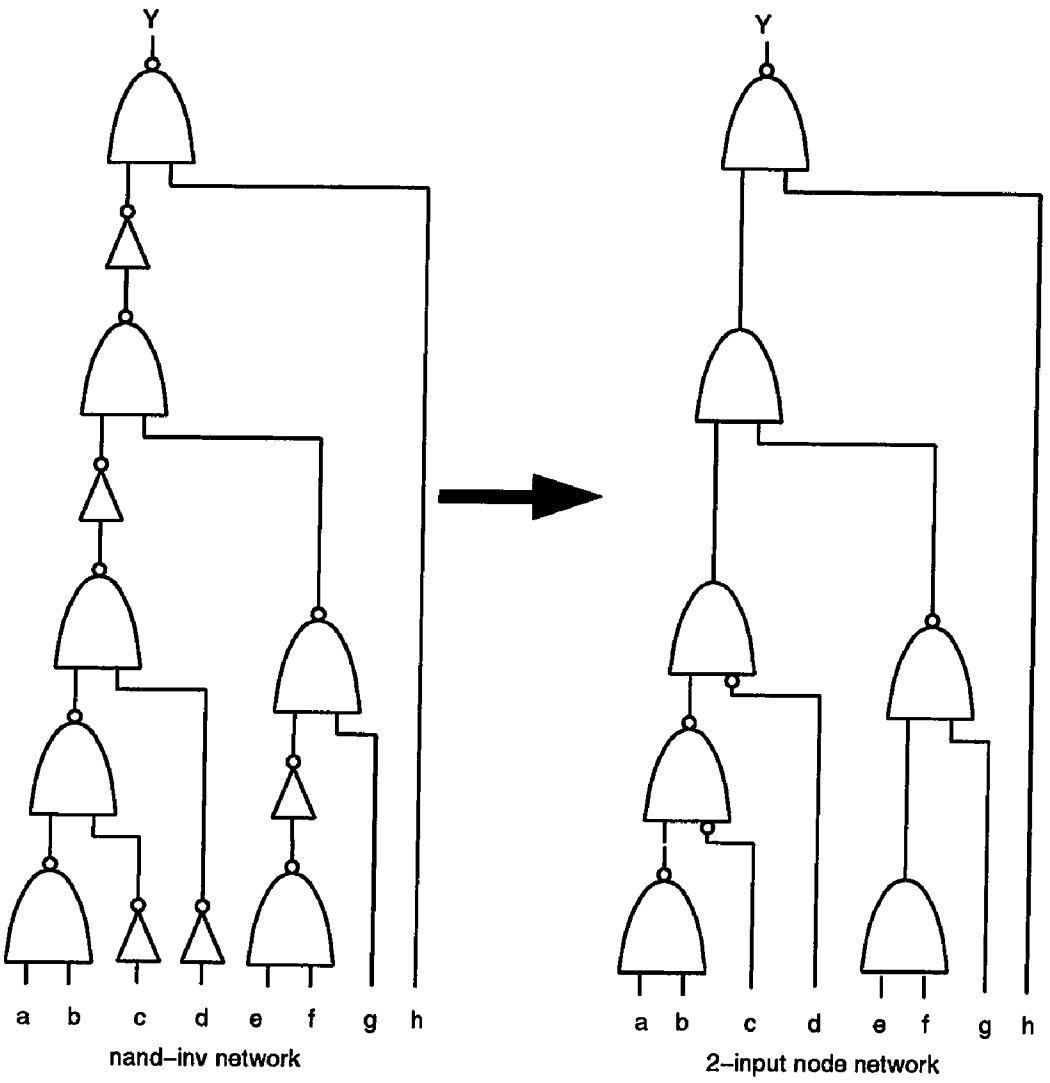


Figure 5.7: The network of Figure 5.3 is converted into a TINN by absorbing the inverters.

5.6.2 Tree covering

We set the cost for match m, at node n, as

$$cost(m) = w \times area\_cost(m) + (1 - w) \times delay\_cost(m).$$

The *area_cost* is computed as

$$area\_cost(m) = 1 + \sum_{v_i \in inputs(m)} area\_cost(v_i),$$

where the '1' represents the cost of a single LUT. The *delay_cost* is computed as

$$delay\_cost(m) = d_{LUT} + \gamma \times \#fanouts(n) + \max_{v_i \in inputs(m)}(delay\_cost(v_i)),$$

where γ represents the delay per fanout and d_{LUT} represents the delay through the LUT. $\#fanouts(n)$ is the number of fanouts of n.

Using the above cost functions, four different mapping solutions are possible:

1. Performance-driven: Net delay is taken into consideration by setting an appropriate value for γ.

2. Area-delay trade-off: Stepping through different values of w, solutions with different areas and delays are obtained.

3. Minimum depth: w is set to 0, γ is set to 0 and d_{LUT} is set to 1. This gives a mapping solution with minimal topological depth.

4. Minimum area: w is set to 1. This gives a mapping solution with minimal LUT count.

Figure 5.8 shows how the network of 2-input nodes of Figure 5.7 is covered by the LUT patterns of Figure 5.6.

5.6.3 Handling reconvergent fanout

Consider the DAG of Figure 5.9. Nodes b and c have 2 fanouts each. The fanouts of b reconverge at node Y. Let us assume that we are using a 3-input LUT library (see Figure 5.5). The function at Y can be expressed as a 3-input function with inputs a, b and c, thereby implying that a 3-input LUT pattern with inputs a, b and c should match at node Y. However, using the library patterns of the Section 5.5, we will need a 5-input LUT pattern (see Figure 5.10) to cover the nodes of the 3-input LUT match at node Y.

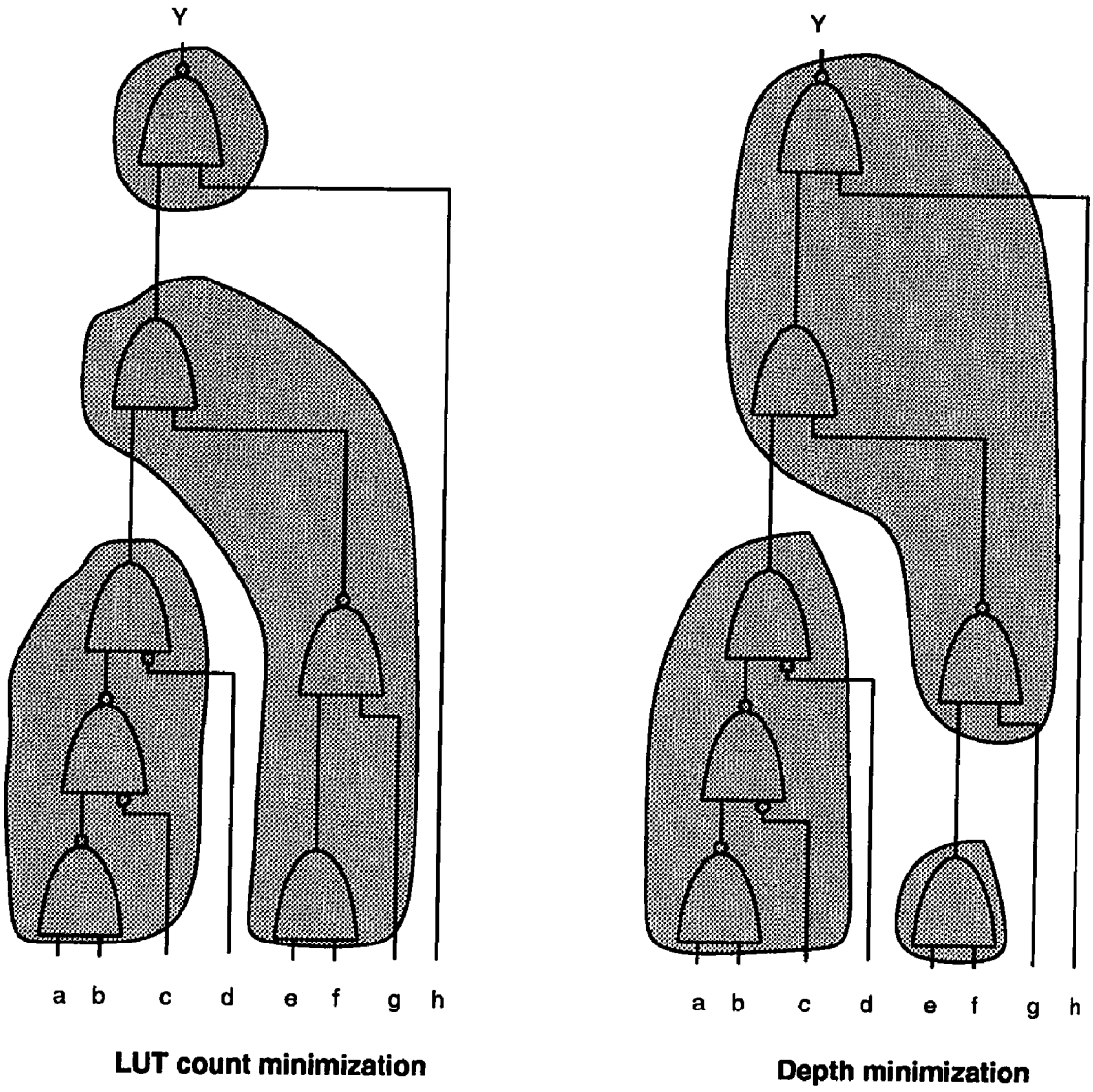


Figure 5.8: 4-input LUT mapping

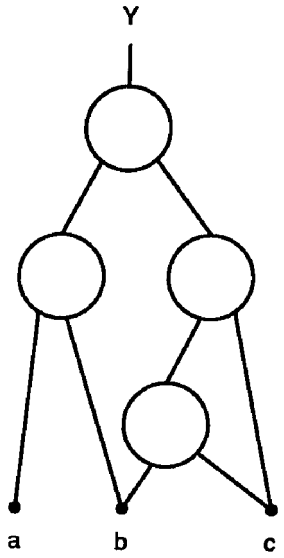


Figure 5.9: A 5-input LUT pattern is required to cover a 3-input LUT match, because of reconverging fanouts.

Unless we use the 5-input LUT pattern, this match at node Y will not be found. This example suggests a general strategy to handle reconverging fanouts. If we are performing a K-input LUT mapping, we must use a $(K+s)$-input LUT library during the matching process. Here, s (> 0) is called the *safety-factor*. For each $(K+i)$-input LUT match, m_{k+i}, where $0 < i \leq s$, the *actual* number of inputs, n_{act}, is counted using the formula

$$n_{act} = K + i - n_{fo}.$$

n_{fo} is the number of fanins that fan out to more than one of the match's inputs. If $n_{act} > K$, the match is discarded. Otherwise, the match is treated as valid and its cost is computed.

Experiments show that values of $s > 5$ do not alter the mapping solution, and hence $s \sim 5$ seems a reasonable value for the safety-factor.

5.7 Fast tree matching

It can be observed that the LUT patterns are very regular. A LUT with K inputs has patterns constructed from patterns of LUTs with x inputs, $x < K$. For example, in

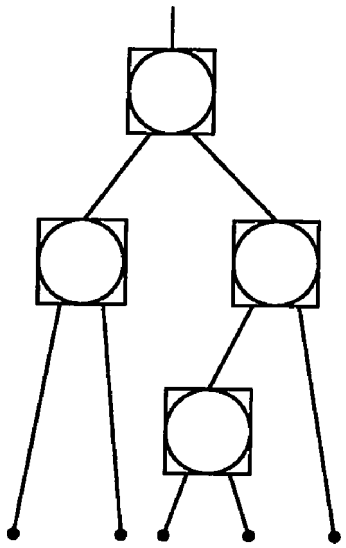


Figure 5.10: This 5-input LUT pattern can cover the nodes in Figure 5.9.

Figure 5.6, the 4-input LUT patterns are constructed from 2-input LUT patterns and 3-input LUT patterns. Using this observation, we can speed up the pattern matching process in the following manner.

5.7.1 Principle

At each tree node, an array of best costs is stored. The array index corresponds to the number of inputs in the matching pattern. At index i, we store the combination [2] of the fanin costs of the best match with i inputs. To find the best K input pattern match at node n, we combine the best x input pattern cost of the left child of n with the best $K - x$ input pattern cost of the right child of n, for $x = 1 \ldots (K - 1)$, and store the pattern with the best cost. The best cost amongst all patterns with i inputs, $1 < i \leq K$ is stored at index 1. The cost at index 1 includes the cost of the matching pattern, whereas the costs at the other indices represents only the fanin costs. For example, consider the case of LUT count minimization. Let N be the node under consideration. If $c(i)$ represents the cost at

[2]The manner in which the fanin costs are to be combined is dependent on what is being optimized. If it is LUT count, then the combination is an arithmetic sum of the fanin costs; if it is delay, then the combining operation chooses the maximum of the fanin delays.

index i, then the minimum fanin cost is given by

$$m = \min_{1<i\leq K}\{c(i)\},$$

then

$$c(1) = m + A_l,$$

where A_l $(= 1)$ is the area of a LUT.

An i-input pattern match requires $(i-1)$ cost computations, and since a K-input LUT mapping requires $(K-1)$ pattern matches, at each node we require

$$\sum_{i=2}^{K}(i-1) = \frac{K(K-1)}{2}$$

cost computations. Thus the number of pattern matches are $O(K^2)$. This means that the effective library size is $O(K^2)$ as a consequence of fast-matching.

5.7.2 Example

The fast matching technique is best illustrated by means of an example. Consider Figure 5.11 where we show the mapping procedure, for the TINN of Figure 5.7. The objective is to obtain a minimum area (LUT count) mapping solution. At each node, an array of best costs is shown. The array index is denoted by i, and runs left to right. The area cost (i.e., LUT count) is shown below each index. All the primary input nodes (i.e., a, b, etc.) have 0 area cost. For example, consider the array at the left of the 2-input NAND gate S. There is only one LUT pattern matching at this gate, and that is the 2-input LUT pattern. The fanins to this pattern are a and b, and each has a fanin cost of 0. Hence, the sum of the fanin costs (since we are minimizing area, the combination is a sum) is 0, and this is stored at index 2. Assuming all LUTs have the same area cost of 1, the best cost at S, stored at index 1, is $= 0 + 1 = 1$.

As a more detailed example, consider the computation of the best 4-input LUT pattern match at node N in Figure 5.11. The following costs have to be first computed.

1. C_{13}, corresponding to the left child's 1-input and right child's 3-input costs, as shown in Figure 5.12.

2. C_{22}, corresponding to the left child's 2-input and right child's 2-input costs, as shown in Figure 5.13.

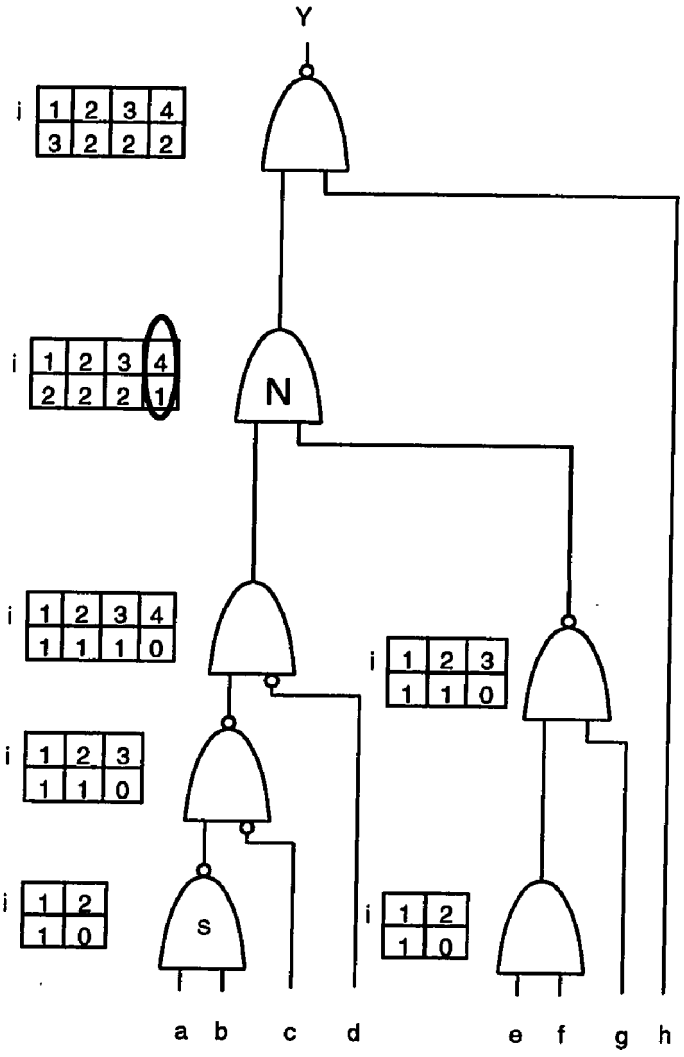


Figure 5.11: Example to illustrate fast tree-matching

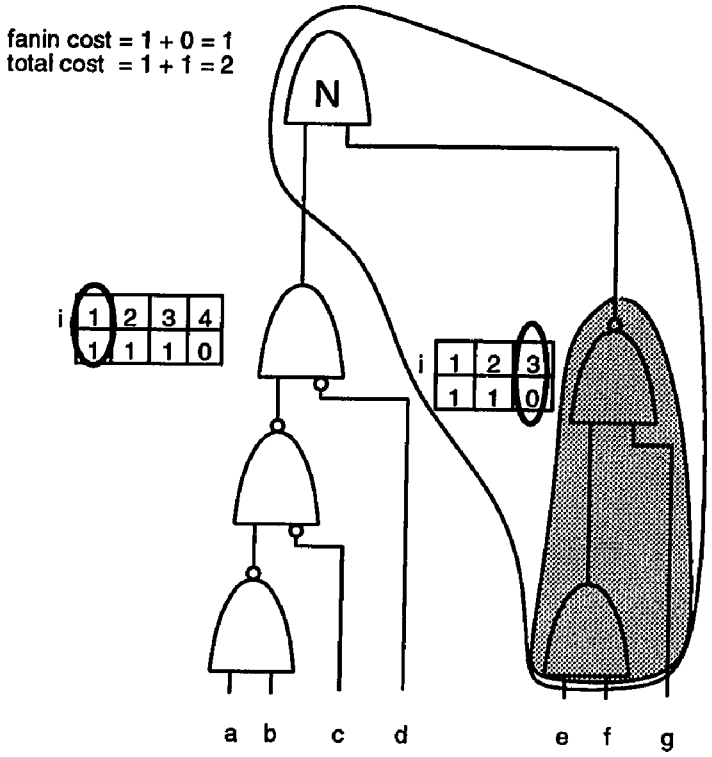


Figure 5.12: Computing cost C_{13}.

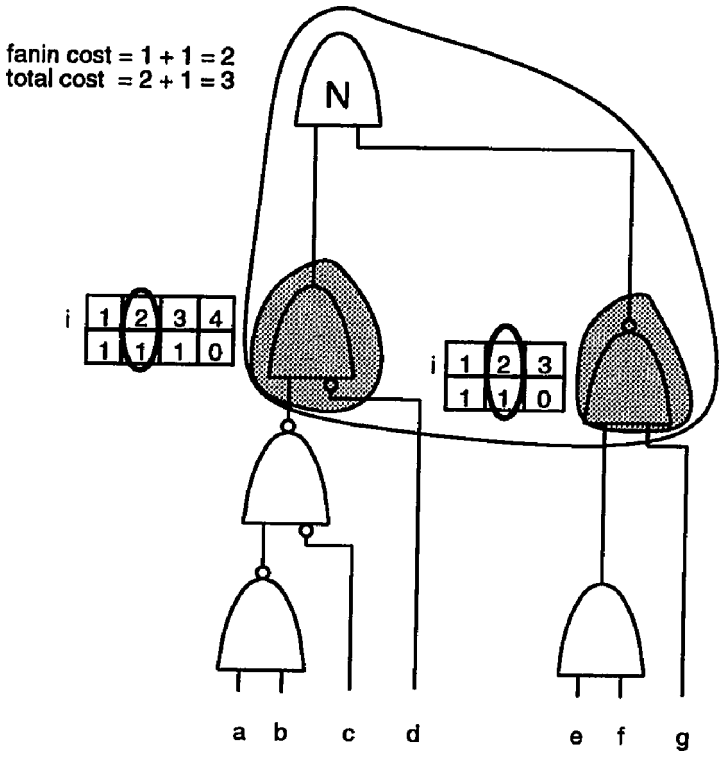


Figure 5.13: Computing cost C_{22}.

3. C_{31}, corresponding to the left child's 3-input and right child's 1-input costs, as shown in Figure 5.14.

Each cost computation involves just two memory look-ups and one addition, and is hence computationally inexpensive. The minimum of C_{13}, C_{22} and C_{31} is stored as the fanin cost at index 4, for the array at node N in Figure 5.11.

5.7.3 Drawbacks

The fast tree matching approach gives optimal results when the subject graph is a tree, with leaves having only one fanout each. If the leaves have more than one fanout, then the approach can give sub-optimal results. Consider the mapping shown in Figure 5.15(a). The tree shown is a part of a larger circuit. The best costs, for area minimization, are shown in parentheses at each node. Nodes a, b, e and f are multi-fanout nodes, and we assume

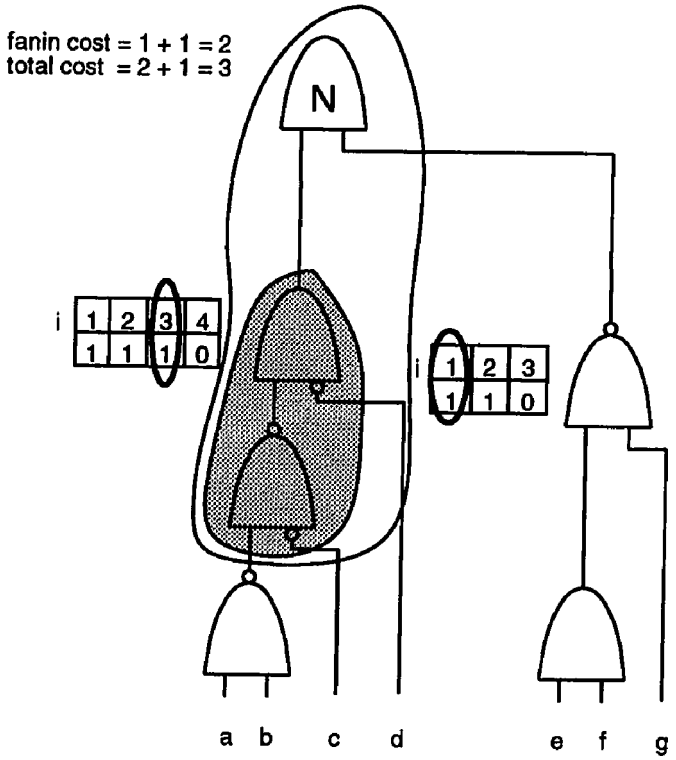


Figure 5.14: Computing cost C_{31}.

that the DAG has been partitioned into trees by breaking at such multi-fanout nodes (see Figure 5.4).

In Figure 5.15(a), consider the 3-input LUT matching at node w. Two matches are possible, as shown by the ovals in the figure. However, the shaded oval has lower cost (5) compared to the unshaded oval (6). Hence the shaded oval is stored as the best 3-input match, at array index 3 at node w. However, this locally optimal solution (locally at w, that is), is not globally optimal. Consider node x, where the multiple fanouts of node b reconverge. Let us consider a 4-input LUT matching at node x. Three possible 4-input LUT matches are possible, as shown in Figures 5.15(b), 5.15(c) and 5.15(d). Figure 5.15(d) has the least area cost; but the fast matching approach cannot find this match. This is because the best match is obtained by combining the unshaded oval match at node w with the 2-input match at node p. However, fast match stores only a single match at each index. Since the unshaded oval was discarded in favor of the shaded oval at node w, the best 4-input LUT match at node x is lost.

5.8 Experimental results

The **dpmap** algorithm has been implemented in the SIS framework. Experimental results are reported in this section. We have compared our performance-driven mapping results with **mis_pga**. Area-delay results are also reported. In addition, for sake of comparison with prior work, we also ran the mapper with depth minimization and area minimization options.

For all our experiments, we started with a network optimized by standard methods [Saldanha 89]. We decomposed the input network into a network of 2-input nodes, using a 2-input library (the second approach described in the Section 5.6.1). The network was then partitioned into a forest of trees and the tree-covering algorithm was used on each such tree. We tried both techniques of partitioning a DAG into trees (as described in Section 5.3.2), and we have reported the better of the two results (since **dpmap** is fast, it is easy to try both techniques). We performed a 5-input LUT mapping (i.e., $K = 5$) and used fast tree matching. To account for reconvergent fanouts, we set the safety factor, s, to 5. Our MIS script is: *read_library 2ip.genlib, map, dpmap.* (*dpmap* performs the LUT mapping). **dpmap** is fast, and in general takes less than a minute of CPU time, on a DEC 5000.

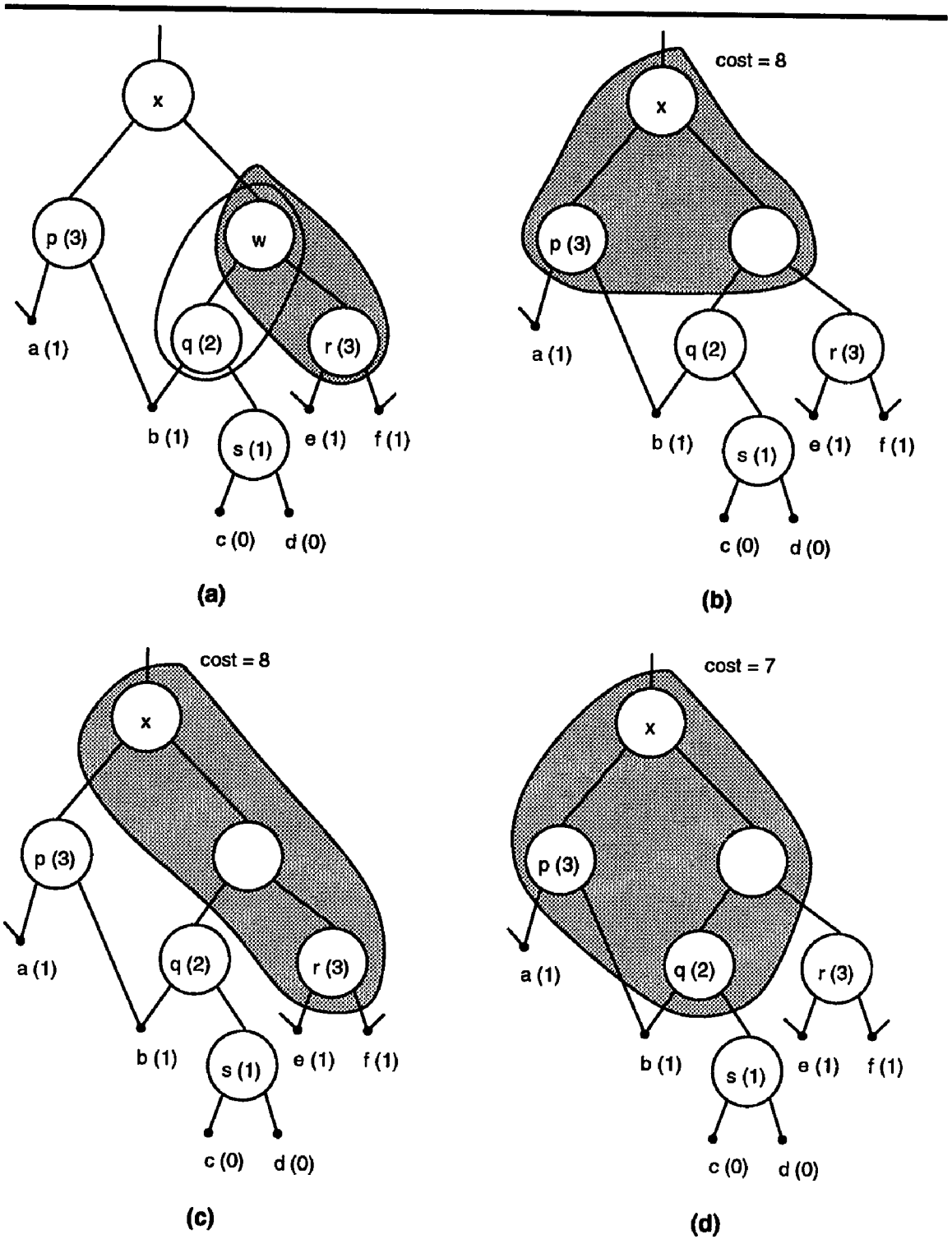


Figure 5.15: Example to illustrate the fast matching drawback

5.8.1 Performance-driven mapping

In this experiment, the net delays are incorporated into the cost computation by using the delay model of Chapter 2. The mapping solutions were placed and routed on the Xilinx XC3090 [Xilinx 89], using Xilinx's **apr** software and a delay analysis was performed using Xilinx's **XDelay** timing analyzer. Table 5.5 shows the delays for some MCNC examples (we have shown only examples with delays more than 100 ns, and those that could fit into the 3090). Our results are compared with **mis_pga**. It was found (by trial and error) that a γ of 1.7 ns/fanout gives a good estimate of the net delay, when the number of logic blocks is less than 60% of the maximum. The Xilinx part used had a CLB delay of 5.5 ns, and hence d_{LUT} was set to 5.5 ns.

Example	**dpmap** delay (ns)	**mis_pga** delay (in ns)
sao2	104.9	96.6
alu4	187.2	207.3
duke2	132.1	176.0
C499	118.8	216.8
apex2	104.0	124.0

Table 5.5: Performance-driven mapping results of **dpmap**. Delays are tabulated after place and route.

From the table, it can be observed that **dpmap** does significantly better, in all examples except sao2. A single value of γ has been used for all the benchmarks. This value may not be suitable for some examples, and sao2 is one such case.

5.8.2 Area-delay trade-off

LBM provides a framework to do effective area-delay trade-offs. In Figure 5.16, we see that by varying the value of w, the relative weights of the area and delay costs (see Section 5.6.2), **dpmap** generates solutions with different number of LUTs and levels. The experiment was conducted with $\gamma = 0$ and $d_{LUT} = 1$, as in the case of the minimum depth experiment.

5.8.3 Minimum depth

To put this work in perspective with published techniques, we compare the minimum depth mapping solution of **dpmap** with **flowmap**[Cong 92] in Table 5.6. **flowmap**

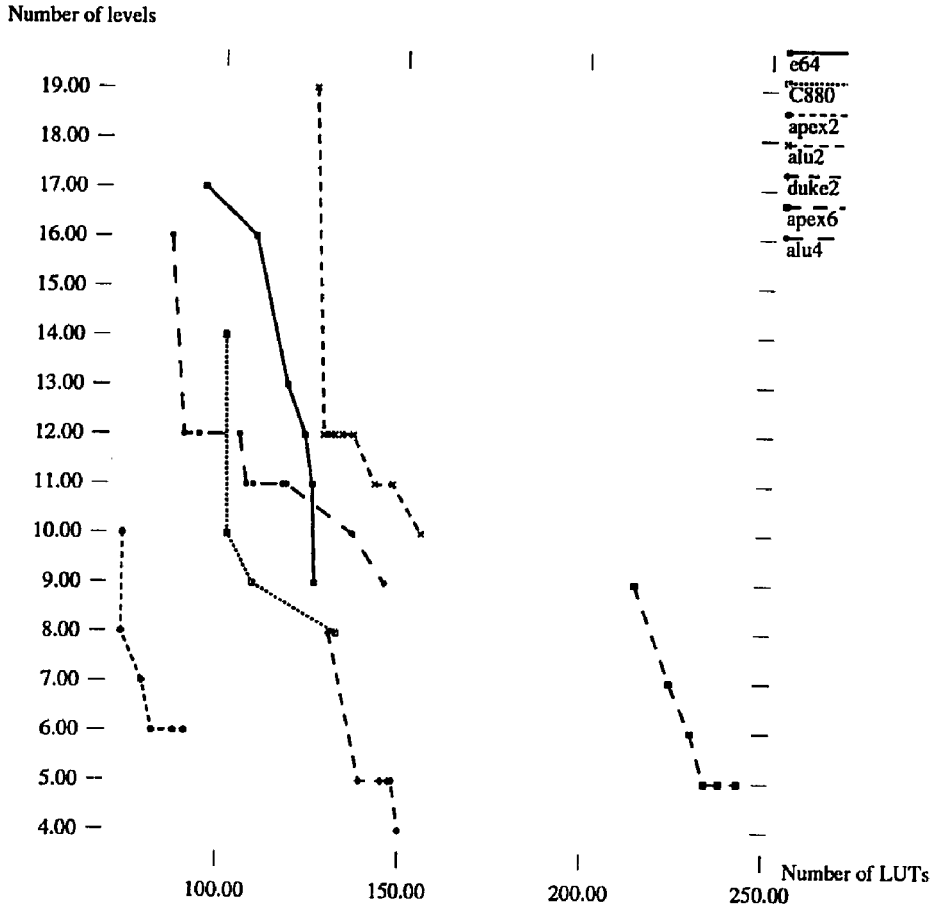


Figure 5.16: Area-delay trade-off.

Example	dpmap		flowmap			
			2ip lib		dmig	
	LUT	dpt	LUT	dpt	LUT	dpt
z4ml	8	3	7	2	5	2
misex1	14	3	18	3	19	3
vg2	34	3	33	3	52	4
count	40	5	38	5	43	5
9symml	51	4	63	4	96	5
9sym	69	5	88	5	148	5
apex7	78	4	91	4	100	4
rd84	38	4	54	4	70	4
e64	126	9	132	9	132	9
C880	132	8	152	8	146	8
apex2	91	6	112	6	192	5
alu2	155	10	175	10	168	9
duke2	150	4	203	4	203	4
C499	74	4	78	4	78	4
apex6	243	5	305	5	397	6
alu4	145	9	142	9	253	9
sao2	47	6	42	6	77	7
rd73	25	4	27	4	30	5
misex2	36	3	54	3	53	3
clip	36	5	40	4	52	4
b9	52	3	54	3	92	3

Table 5.6: Minimum-depth mapping solutions. LUTs columns list the number of 5-input LUTs and depth columns list the maximum topological levels.

gives the optimal depth mapping solution for a given input network. **flowmap** was run in the default mode. Before executing **flowmap**, the input network must be *feasible*, i.e., the number of inputs for every node should be less than the LUT inputs. Typically, the number of inputs is restricted to 2. The **flowmap** package has a command *dmig* to convert the input network to a feasible network. However, we found that our 2-input library based conversion yields better results. Both methods are listed in Table 5.6. From the table, we observe that **dpmap** produces the minimum depth solution (comparing with the *2ip lib* column of **flowmap**) in all but 2 cases (`z4ml` and `clip`). The LUT count is lesser by 11% in the case of **dpmap**.

Example	dpmap	mis_pga	chortle-crf	Xmap
z4ml	6	5	7	9
misex1	13	11	11	11
vg2	25	20	21	24
count	43	31	31	31
9symml	46	7	44	55
9sym	65	7	59	73
apex7	67	60	60	65
rd84	37	10	35	36
e64	95	80	80	80
C880	101	82	88	103
apex2	73	67	64	81
alu2	125	109	116	126
duke2	130	110	111	127
C499	81	68	89	75
apex6	214	182	198	231
alu4	86	55	70	98
sao2	33	28	27	37
rd73	21	6	16	21
misex2	34	28	28	28
clip	35	28	31	38
b9	45	39	41	48

Table 5.7: Minimum-area mapping solutions. The table lists the number of 5-input single-output LUTs

5.8.4 Minimum area

Table 5.7 shows the number of 5-input LUTs needed to implement several benchmark examples. Results are compared with **mis_pga**[Murgai 91], **chortle-crf**[Francis 91] and **Xmap**[Karplus 91]. The numbers for these other algorithms are taken from [Murgai 91]. All use the same starting network.

dpmap always does worse than **mis_pga** (**mis_pga** has better area results because it performs logic optimization for LUTs along with mapping), and does worse than **chortle-crf** in all but one example (C499). But it is encouraging to note that our simple covering technique gives comparable results. Different decomposition techniques for the 2-input network generation, and logic optimization techniques for LUTs need to be tried to improve the results.

5.9 Conclusions

In this chapter, we showed how to extend the performance-driven mapping and area-delay trade-off capabilities of standard cell based mappers to LUTs. We introduced the 2-input LUT primitive cell, and used it as a building block to generate a small, practical library. Using a linear delay model, we accounted for net delays during the dynamic programming based mapping algorithm, and hence made the **dpmap** LUT mapper a truly performance-driven mapping algorithm. The library-based approach gives the mapper the ability to be used across a *class* of LUT based FPGA architectures. The special structure of library patterns was exploited to reduce effective library size to $O(K^2)$, for K-input LUT mapping.

Experimental results indicate that the timing delays, after place and route are better than **mis_pga** by about 17.5%. Area-delay trade-offs show that the solutions can span across as many as three Xilinx chips; thus giving the user the choice between area, speed and dollars. Depth minimization results are as good as the **flowmap** results in all but two examples; and **dpmap** has 11% less LUTs.

dpmap suffers from the disadvantage that the solution is dependent on the initial decomposition of the network into 2-input nodes, a problem inherent with LBM techniques. This explains the poor area-minimization results. A second disadvantage is that the LBM approach cannot handle multi-output gates. Most of the LUT based FPGAs have multi-output logic modules (eg., the XC3000 has 2-output LUTs). Mapping for such architectures has to proceed in two steps:

1. Map for single output LUTs
2. Combine the mapped LUTs, using techniques described in [Murgai 91].

This work can be extended to integrate mapping with placement (as in [Pedram 91]) for better net delay estimations. Since LBM is widely used for standard cells, there is a large body of researchers in this field, and any new developments can be easily incorporated into **dpmap**. Libraries for other LUT based architectures, like the Xilinx XC4000 series and the AT&T ORCA need to be developed. The TIL need not be the only primitive cell; number of patterns built from 3-input LUTs are also small, and the mapping using these patterns needs to be investigated.

Part II

Performance-oriented Architecture

Chapter 6

Overview

In this part of the thesis, we focus on the device architecture of FPLDs. In Part I, the question we tried to answer was: *Given a class of devices (viz., LUT FPGAs), how best can we design the mapping software?* In Part II, our question becomes more general: *Given that we need to design a PLS, how best can we design the device?*

In Chapter 1, we listed the desirable features of a PLS, viz., automatic, performance-driven, fast CAD tools and a silicon-efficient, low-delay FPLD. But, CAD and the device are interlinked. We cannot design fully automated CAD if the device architecture is very complex and causes nets to remain unrouted. Nor can it be made performance-driven, if delays in the interconnect are hard to predict. Similarly, the silicon-efficiency of a device depends on how well the resources can be utilized by software aids. It is easy to design an architecture which results in intractable solutions for the software.

Keeping in mind that the fixed quantity of logic and interconnect resources on a device limits the freedom of the software, an architecture that is simple and easy to design with and at the same time utilizing silicon efficiently and having predictable interconnect delays is needed. We further motivate the need for a new architecture in Chapter 7.

Routing is the stumbling block for FPLDs. Device architects have come up with complex routing architectures, as can be seen in the XC4000 and the AT&T ORCA devices. These require intelligent and complex routing software. And to ensure that routing completion does not become a problem, abundant routing tracks are being provided in the newer devices, resulting in silicon wastage. We design our new architecture, *Dharma*, with the intent of keeping routing simple. The simplest way is to provide *full* connectivity, allowing any logic module to be connected to any other. Of course, if we bluntly apply this idea, the

device will end up having nothing but routing! To make this full connectivity possible, it should be possible to time-multiplex the routing hardware. In Chapter 8, we introduce the concept of *levelize-and-hold*, and show how interconnect resources can be time shared. The idea is to implement the multi-level combinational part of the circuit in a folded-pipeline manner, re-configuring logic and interconnection in real time. The *Dharma* architecture is then described in the same chapter.

To illustrate the *Dharma* architecture better, three example circuit implementations are shown in Chapter 9; one sequential and two combinational circuit examples are shown.

As brought out in Chapter 1, the idea of real time reconfiguration can foster families of FPLDs. Several variants of the basic architecture are possible, and some of these are described in Chapter 10.

Synthesis techniques for *Dharma* are then described in Chapter 11. We consider only physical synthesis. Given a net-list, we need to allocate the logic modules to different time slices, in a manner such that on-chip resources are not exceeded. We show that this problem can be formulated as an integer programming problem. We also show a modified Kernighan-Lin technique to solve the allocation problem, and present experimental results for this approach.

Chapter 12 presents analysis of the new architecture. Area and timing analyses are performed, and timing equations are presented. Several circuit examples are compared with the XC3000 performance, using the delay equation.

The chapter on "Unexplored terrain" discusses potential ways to further the architecture related work. *Dharma* is a CAD-oriented architecture, the motivation being to capitalize on software strengths, while avoiding software weaknesses. We point out two synthesis problems which, if solved, can result in improved *Dharma* devices.

Chapter 7

Why Another Architecture?

This part of the thesis presents a new architecture for an FPLD. Before we introduce the new architecture, we must examine the need for another architecture. As seen in the FPLD classification tree (ignoring the shaded portion), Figure 1.3, there are already many different kinds of FPLDs, and one may have the erroneous impression that there are enough devices out there already. In this chapter, we bring out the shortcomings of these devices, justify the need for a better device, and also list the features that are desirable in an FPLD. The next chapter will then present the new architecture that meets these desired features.

As described in Section 1.3, the programmable device forms a part of the PLS, and for it to become a viable alternative to other ASIC design styles, the PLS must have capabilities to meet high performance application requirements.

But, if we examine the history of FPLD evolution, we observe that these devices started off as a means for fast prototyping, essentially to verify the design's logic function correctness, with no regard to timing correctness. Over the years many different architecture styles have cropped up, with increased complexity, and more features, but without any real consideration for the overall PLS improvement or performance improvement.

Let us concentrate on FPGAs, because at the present time only they have the capability, albeit limited, to meet the high-performance ASIC market requirements. PLDs are not complex enough to implement large circuits.

Designing circuits on FPGAs has proceeded very much along the lines of MPGAs (see Figure 1.4):

1. Map the circuit gates into the logic blocks of the FPGA chip.

2. Place the logic blocks on the FPGA plane.

3. Route the connections using the FPGA programmable routing resources.

If performance (clock speed of the FPGA-implemented circuit) is made a criterion, at present there is no commercial FPGA chip and accompanying CAD tool which can guarantee a given clock speed, or perform the above 3 steps while taking into consideration both logic and routing delays. As a result, it is not yet possible to fully utilize a programmable chip in high-performance designs.

Three kinds of drawbacks arise in current FPGAs [1] which use these steps in the design process. We discuss them under the subheadings of Routing issues, Timing issues and CAD issues.

7.1 Routing issues

As outlined in Section 4.1, the programmable interconnection in current FPGAs presents two kinds of problems, routing completion (discussed in this section) and dominant timing delay (discussed in the next section).

Two approaches have come up to tackle the routing completion problem:

1. Architecture approach

2. CAD approach

7.1.1 Architecture approach

In the architecture approach, the belief is that the routing completion is not a real problem, and it arose simply because there were inadequate routing resources on the chip. Therefore, the solution is to increase the routing resources and routing flexibility. Consider the Xilinx FPGA chips. Designers complain of routing completion in the XC3000, but not in the case of the XC2000 or the XC4000. When XC3000 evolved out of the XC2000, logic modules were made more complex, and the number of modules per chip was increased, but the routing structure was left unchanged. However, the latest XC4000

[1] We concentrate on LUT FPGAs, since these are the most popular (in terms of number of units sold).

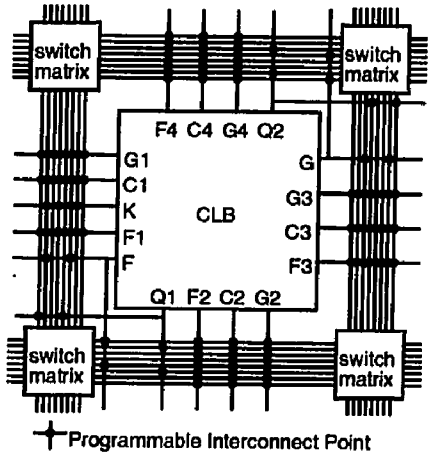
chips have significantly greater routing resources. Compared to the previous families, the XC4000 has twice as many horizontal and vertical *Long Lines* that can carry signals across the length or width of the chip; and the number of globally distributed clock signals has been increased from two to four. Compared to the XC3000 family, the XC4000 family has more than double the routing resources, and they are arranged in a more regular fashion [Alfke, Hsieh 90]. Figure 7.1 (from [Hsieh 90]) shows the routing flexibility and the four kinds of routing resources, viz., single lines, double lines, long lines and switch matrix, of the XC4000 family.

Similarly, the AT&T ORCA series of chips also have an abundance of routing resources [ORCA 93]. ORCA routing resources are made from metal segments called resource routing nodes (R-nodes). These R-nodes are connected together at configurable interconnect points (CIPs) to form user-defined nets. There are two types of R-nodes: internal and inter-PLC (programmable logic cell). The internal R-nodes are used to route signals within the PLC. Internal R-nodes consist of programmable function unit (PFU) input R-nodes, PFU output R-nodes, switching R-nodes, and bidirectional R-nodes. Inter-PLC R-nodes are used to route signals from one PLC to another. These are known as x1 R-nodes, x4 R-nodes, xL R-nodes, and direct R-nodes (see Figure 7.2). The x1 R-nodes are one PLC long, the x4 R-nodes are 4 PLCs long, and the xL R-nodes span the entire length or width of the ORCA chip. The direct R-nodes allow high-speed connections to directly adjacent PFUs on all four sides without using inter-PLC routing resources.

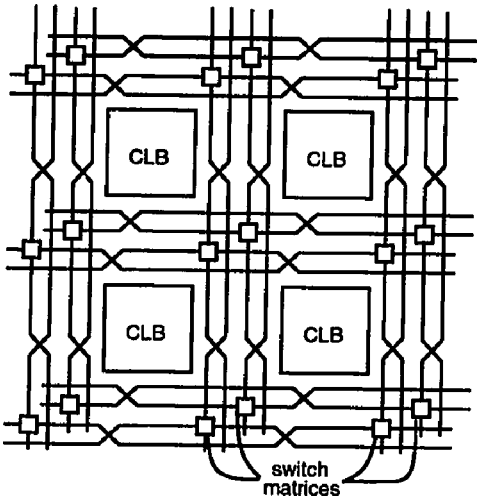
From Figures 7.1 and 7.2, we see that the architectural approach favors increase in the routing resources and routing flexibility. This eases the task of the router, and routing completion is usually possible. However, silicon resource has been wasted, and because of larger number of switches per track, the parasitic capacitance on the track is quite large, and hence causes poor timing performance.

7.1.2 CAD approach

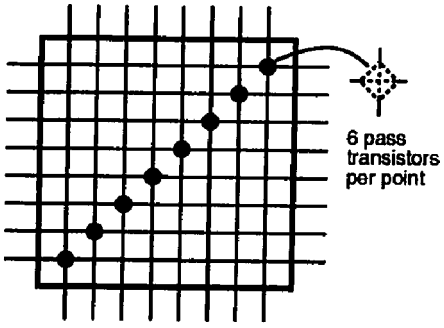
In the CAD approach, the CAD tool designer assumes that the problem with routing is because of ineffective CAD tools, and hopes to rectify the problem by modifying the CAD tool. One such CAD approach has been discussed in Chapter 4. Other approaches [Schlag 92, Trimberger 92] also use a similar principle, and try to modify the synthesis solution to alleviate the problem of incomplete routing. All these approaches have met with



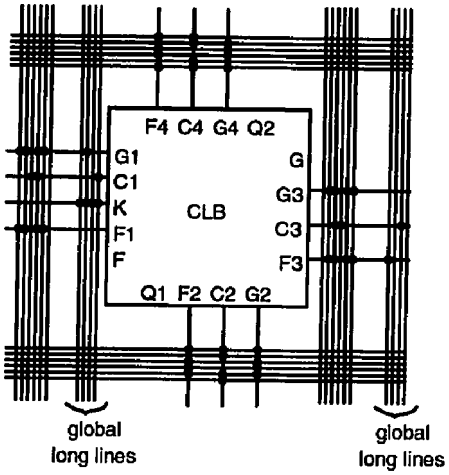
(a) CLB connections to single-length lines



(b) Double-length lines



(c) Switch Matrix connections



(d) Long Line resources

Figure 7.1: XC4000 family routing resources. This family has more than twice the routing resources of its predecessor, the XC3000 family.

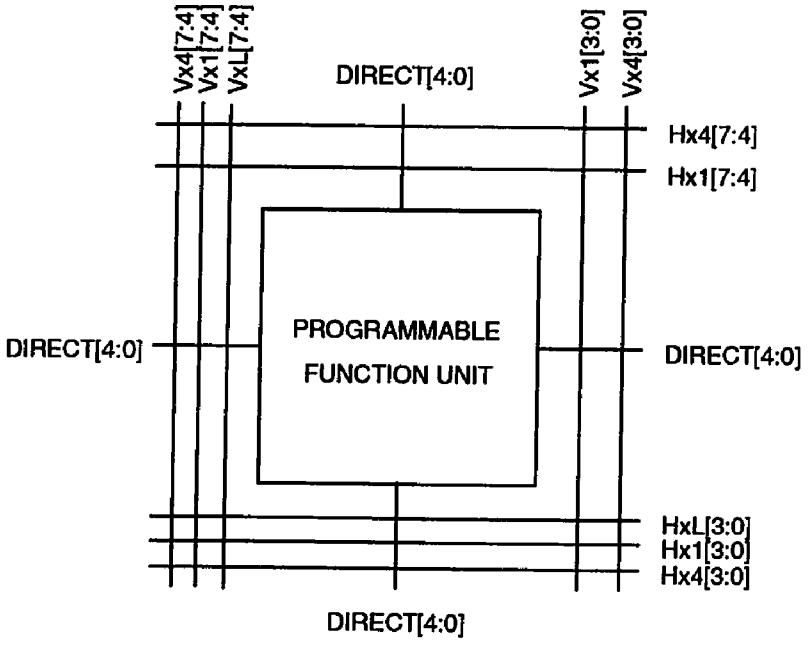


Figure 7.2: AT&T ORCA family routing resources. Figure shows the inter-PLC R-nodes.

limited success. There is no single quantity that can be effectively used as a routability measure, to determine apriori (i.e., before complete place and route) the routing characteristic of a synthesized solution, and having a variety of routing resources only increases this problem.

7.1.3 Summary

To sum up, routing completion is a problem in FPLDs. The architecture approach hopes to solve this problem by increasing the number of routing resources, but this could result in silicon wastage and increased parasitic capacitance. The CAD approach tries to modify the mapping solution to improve routability, but it has met with only limited success.

7.2 Timing issues

The second issue of concern with current LUT based FPGA devices is the propagation delay in the interconnections. This concern involves two factors.

1. Interconnection delay is the dominant delay.

2. Interconnection delay is difficult to estimate.

Let us study each in turn.

7.2.1 Dominant interconnect delay

We performed an experiment to study the interconnection (or routing) delay. We used the Xilinx XC3000 series chips as a test bed. Several MCNC combinational benchmark circuits were placed and routed on these chips. These circuits were mapped to Xilinx using the **mis_pga** mapping technique [Murgai 91a][2]. The mapper generates an output circuit with 1-output LUTs, each having ≤ 5 inputs. We assign one such LUT to each CLB on the Xilinx chip. Each benchmark circuit is placed and routed on the smallest possible XC3000 series chip (i.e., the smallest chip which satisfied the condition, # CLBs $\geq$ # LUTs). However, if there were unrouted nets on such a chip, the next larger chip is used.

[2]It is possible that the results obtained could be different if a different mapping is used. However, the objective here is to just get a feel for the routing delay. We are not interested in obtaining the best possible answer.

Example	LUTs	Levels	CLB delay (ns), C	Routing delay (ns), R	R/C	Xilinx Part	CLBs available
5xp1	21	2	16	38.8	2.4	XC3020	64
C499	199	8	64	249.3	3.9	XC3064	224
C880	259	9	72	237.9	3.3	XC3090	320
alu2	121	6	48	106.4	2.2	XC3042	144
alu4	155	11	88	195.5	2.2	XC3064	224
apex7	96	4	32	103.6	3.2	XC3042	144
b9	49	3	24	50.2	2.1	XC3020	64
bw	28	1	8	34.6	4.3	XC3020	64
clip	54	4	32	55.5	1.7	XC3020	64
count	81	4	32	68.1	2.1	XC3030	100
duke2	164	6	48	114.2	2.4	XC3064	224
e64	213	5	40	130.5	3.3	XC3090	320
f51m	23	4	32	44.3	1.4	XC3020	64
misex2	37	3	24	36.3	1.5	XC3020	64
sao2	46	5	40	52.4	1.3	XC3020	64
vg2	100	8	64	54.9	0.9	XC3042	144
Average					2.4		

Table 7.1: Results from an experiment demonstrates that the routing delay is, on an average, 2.4 times the logic delay, for the XC3000 series.

The results of this experiment are tabulated in Table 7.1. The second column lists the number of 1-output LUTs in the examples, and the third column lists the number of levels of logic on the path with maximum delay. All Xilinx parts were of speed grade '70' (i.e., maximum speed of operation is 70MHz), and the CLB delay is 8ns for such parts. Hence, the CLB delay, C, in column 4, is obtained by multiplying the corresponding numbers in column 3 by 8. After place and route, using Xilinx's **apr** software, timing analysis was done using Xilinx's **XDelay** timing analyzer. Column 5, in Table 7.1 lists the worst-case routing delays, R, obtained from **XDelay**. We compute the ratio of routing to logic delays, R/C, and this is tabulated in column 6. Column 7 lists the part name of the Xilinx part, on which the example circuit was placed and routed and column 8 lists the number of available CLBs in that part.

From the R/C column, we see that the interconnect delay is significantly larger than the logic delay. Over the 16 examples listed in Table 7.1, the routing delay is, on the average, 2.4 times greater than the logic delay.

7.2.2 Widely varying interconnect delay

As shown in Section 5.1, the interconnect delay varies widely from circuit to circuit. For circuits with the same number of logic levels, the routing delay varies by a factor of about 60%. Our performance driven mapping algorithm, **dpmap**, tries to model the interconnection delay, using the equation $d(N) \propto s(N)$, where $d(N)$ is the delay on net N, and $s(N)$ is the number of sinks on net N. However, there is no single constant of proportionality that works well. For the XC3000 series, the value 1.7 was found to work well only when the chip was less than 60% populated. This wide variation is due to the nature of the programmable interconnect resources. If two points on the chip, close to each other, need to be connected, and if the interconnect resources in the intervening space have been used up, then the connection will have to be made in a round-about fashion, resulting in a very different net delay.

7.3 CAD issues

The 3 steps, on page 81 are the responsibility of the CAD tools accompanying the architecture. In case of failure to achieve satisfied routing the user may be required to change placement and routing manually which is both tedious and time consuming.

Steps 2 and 3 present two kinds of problems. Firstly, placement and routing tools perform badly for FPGAs, when timing specifications need to be considered. Although there have been new advances in timing-driven placement and routing techniques for standard cell and gate array design styles, these techniques have not yet been successful for FPGAs. This is partly because the synthesis procedure is unable to make an accurate estimate of net delays, and partly because the programmable routing structure makes it difficult for the placement tools to predict the router's behavior. Secondly, placement and routing tools are time consuming, and constitute the main bottleneck in the programmable logic system's design cycle time reduction.

In applications where the programmable device is used for prototyping, a different problem arises. Whenever small changes to the design are made (either specification modification or correcting errors), the steps have to be repeated. Not only does this process take time, but the new solution produced after place and route may be completely different from the previous solution. From performance standpoint, this new solution could very well invalidate the small change, and this means that additional design iterations will be required to obtain the desired result. This can be quite frustrating to the designer.

7.4 Desired features

Keeping the above three issues in mind, we list the features desired in any FPLD architecture.

1. Guaranteed routability. The programmable interconnect architecture should be such that a given net can easily be routed.

2. Fast interconnections. Guaranteed routability may imply that there should be an abundance of routing resources. However, for the FPLD to be useful in applications with performance constraints, the routing architecture should be such that the delay in the nets is as low as possible.

3. CAD friendly. With the overall PLS in mind, the architecture should be amenable, from the CAD point of view. This means that the routing delays should be easy to estimate, and that the CAD tools should be able to perform their function in a fast manner.

In other words, the chip architect has to acknowledge that certain problems are too difficult to solve for the CAD tool, and hence design the programmable chip such that these problems do not occur. Research in the area of combining architecture and CAD is ongoing in the universities [Tseng 92, Brown 92]; but the efforts therein have been to modify existing FPGA architectures.

7.5 New architecture

We present an entirely different architecture. It has programmable routing elements and programmable logic elements like current FPGA chips. However, the architecture has been designed such that steps 2 and 3 on page 81 are reduced to almost trivial steps, and the CAD tool's responsibility now lies only in performing step 1. This means that the mapping solution in itself decides the performance of the circuit. The new architecture, named *Dharma* is presented in the next chapter.

Chapter 8

New Architecture

This chapter describes the new architecture, *Dharma*. The chapter is divided into three sections. In the first section, we explain the concept of levelize and hold, the central principle on which *Dharma* hinges. The next section explains the architecture itself. The last section describes the procedure involved in *Dharma's* internal operation.

8.1 Principle of levelize and hold

In this section, we develop the central idea on which *Dharma* is based. Our idea is to time-share routing resources. We start by describing signal propagation in logic circuits, and then show how levelization and the addition of latches can make it possible to time-share routing.

8.1.1 Signal propagation in logic circuits

In Figure 8.1 we show the general model of a sequential logic circuit. It consists of a combinational logic block and sequential elements(latches). The output of the latches are fed as input to the combinational logic block, and some outputs of the combinational logic block are fed back as inputs to the latches. In addition to the outputs from the latches, the combinational block also gets inputs directly; and some of its outputs need not feedback to the latches. The speed of this circuit is defined as the rate at which the latches can be clocked. For correct operation, if T_c is the time-period of the clock, D the delay through the combinational block and T_{sd} the latch set-up plus latch delay times,

$$T_c \geq D + T_{sd}.$$

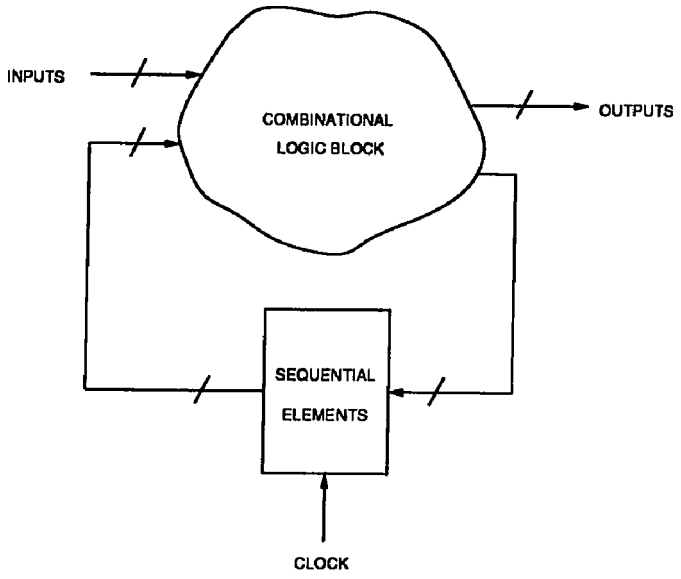


Figure 8.1: A general model of a sequential logic circuit.

What do we mean when we say that the combinational block has delay D? Let the inputs to the combinational block assume their correct value at time instant t_i. Then, if the time instant at which *all* outputs have assumed their correct values (as determined by the function the combinational block is implementing) is t_o, then $D = t_o - t_i$. Some of the outputs may assume their correct values earlier than the other outputs, but in the definition of the delay D, we require *all* the outputs to have settled to their correct values.

How do signals flow in such a circuit? Assume that at some instant of time t_i, the latch outputs and other inputs to the combinational block are ready. After D units of time, the outputs of the combinational block are valid, and after another T_s (latch set-up time) units of time, the values can be latched. After T_d (latch propagation delay) units of time, the latch outputs are valid, and the signal flow cycle repeats.

Let us now observe how signals propagate through the combinational block. When the number of inputs and outputs are large, the combinational block is typically implemented as a multi-level circuit, as shown in Figure 8.2.

The nodes of the directed acyclic graph (DAG) in Figure 8.2 represent single output combinational gates, and the edges represent the interconnection between them.

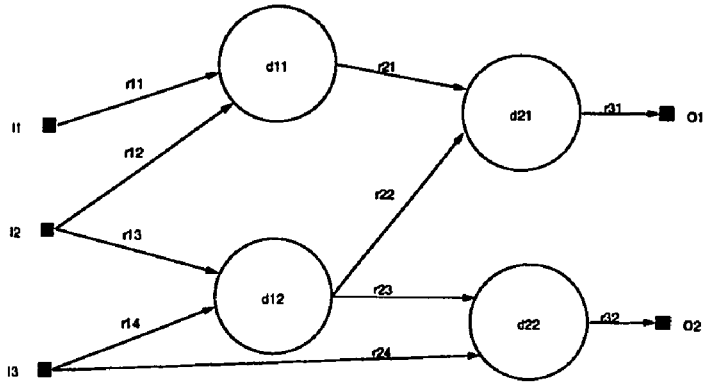


Figure 8.2: Multi-level implementation of combinational logic block.

The symbols within the nodes represent the delay through the gates and the interconnection delay is marked by symbols on the edges.

For ease of explanation, and without loss of generality, let us assume the gate delay to be the same for all gates, and let it be δ. Similarly, let us assume the interconnect delay to be the same for every source-sink pair, and let it be γ. With these assumptions, we can immediately see that the delay of the entire combinational block, D is given by

$$D = L\delta + (L+1)\gamma,$$

where L is the maximum number of levels in the DAG.

In Figure 8.3 we show how the signals propagate with time. In the first time interval γ, the signals propagate through the first level of interconnect, following which they propagate through the first level of logic in δ time units. Hence, at the end of $\gamma + \delta$ units of time, after inputs are valid, the signals have reached the outputs of the first level of logic. Similarly, the signals propagate through the other levels of logic and finally reach the outputs of the Lth level after time $L(\delta + \gamma)$. One additional γ time interval is required to propagate through the interconnection to the output (or latch inputs).

Consider Figure 8.4, where we have introduced latches at all the inputs and at the outputs of all the nodes of Figure 8.2. For the present, let us assume that these latches are ideal, and have no set-up/hold times, nor any propagation delays. Let the inputs be latched at time instant t_i. After time interval $\gamma + \delta$, the outputs of the first level of logic are latched. At time instant $t_i + 2(\gamma + \delta)$, the outputs of the second level of logic are latched,

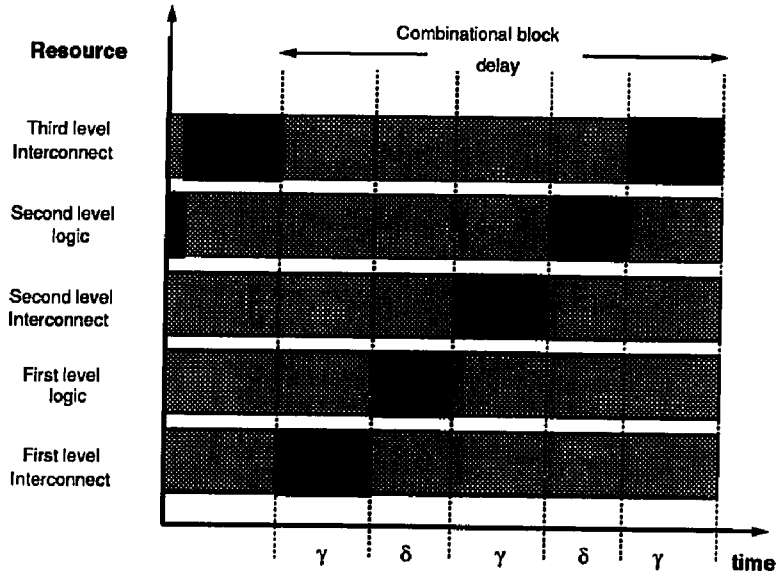


Figure 8.3: Signal propagation in Figure 8.2 and the identification of 'idle' resources. Lightly shaded regions correspond to time-intervals during which the logic or interconnect resource is not in use, when ideal latches are used to hold output logic levels, as illustrated in Figure 8.4.

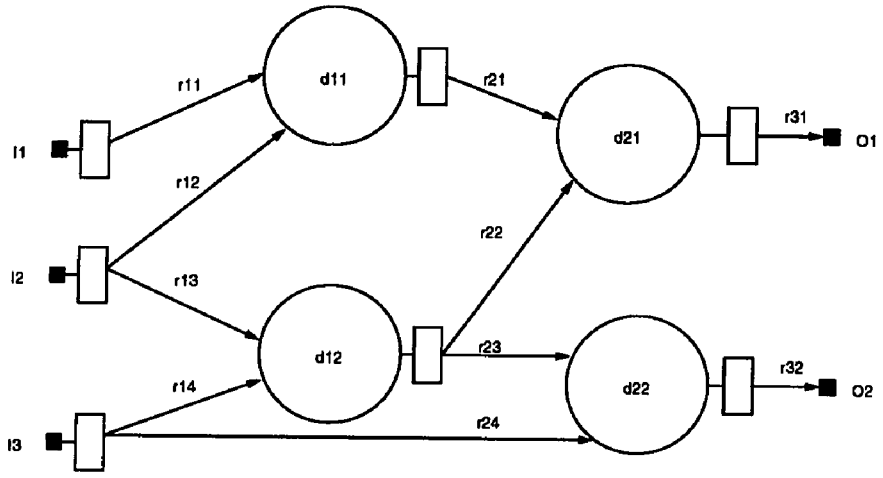


Figure 8.4: The DAG of Figure 8.2 is redrawn here with ideal latches at the outputs of every node and also at the inputs to the combinational logic block.

and so on. Since the latches make available the signal value to higher levels of logic, the interconnect and logic resources at level i perform useful work only during the time interval $t_i + (i-1)(\delta+\gamma)$ to $t_i + i(\delta+\gamma)$ and are 'idle' (not producing anything new) for the rest of the time during a clock cycle t_i to $(t_i + D)$.

In Figure 8.3, we have used 2 different shades. Each resource, be it interconnect or logic, is shaded black when signals are propagating through it for the first time. At the end of the black shaded region, signals are available for use by the next resource (logic or interconnect, as the case may be). The light (grey) shade is used to show that the resource is, in some sense, 'idle', since it is not producing anything new. Our aim is to reuse the resource during its idle period.

8.1.2 Levelize-and-hold

From the above observation, we can develop a strategy for time-sharing resources. We first perform a topological levelization of the DAG representing the combinational block of the input circuit. Nodes at the same level cannot be time-shared since they evaluate their outputs in parallel. Nodes at different levels can be time-shared. i.e., nodes at level $i+1$ can use the same logic resource used by nodes at level i, since the nodes at level $i+1$ cannot

start their function evaluation until the nodes at level i have completed their evaluation. Similarly, the interconnect resource connecting level i to $i+1$ can be used to connect level $i+1$ signals to level $i+2$.

Hence, *Dharma* needs to have one level of interconnect, one level of logic, and latches to hold the values of signals generated. We also need a means to change the interconnection and logic function, when the next level of logic and interconnect are to be implemented. In the next section we describe how this is achieved in *Dharma.*

8.1.3 Sequential elements

So far we only described how the combinational block of Figure 8.1 can be implemented by time-sharing logic and interconnection. Do the sequential elements of Figure 8.1 pose a problem? No, they do not, since they are required to latch the values of the outputs of the combinational block, and make them available as inputs when the next clock period begins. In our levelize-and-hold scheme, this function is already being performed by the latches of *hold.* Thus, using levelize-and-hold, the only difference between combinational and sequential circuits is that the last level is fed back to the first level in the latter, but not in the former.

8.2 *Dharma* Architecture

In this section, we describe the new architecture. We first give an overview using a high-level block diagram and then look at the detailed block diagram. The sub-blocks are then explained in turn.

8.2.1 High-level block diagram

The central idea of *Dharma* is the time-sharing of silicon area, especially the interconnection resources. This is made possible because of the topological levelization of the input circuit being implemented on *Dharma. Dharma* consists of four main blocks (see Figure 8.5):

1. A one-dimensional array of dynamic logic modules (DLM)

2. A one-dimensional array of pass-buffers

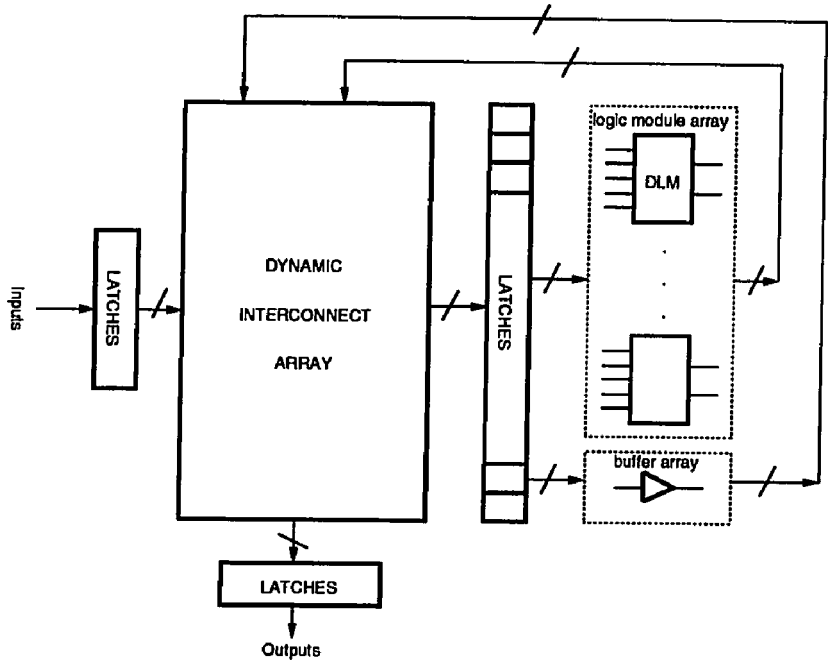


Figure 8.5: High level block diagram of *Dharma*

3. Dynamic Interconnection Array (DIA)

4. Latches

A circuit to be implemented on *Dharma* is first levelized. Let l be the number of levels. Then the implementation of this circuit on *Dharma* is a cyclic repetition of l internal cycles. In internal cycle i, the logic modules are configured to implement the functions in level i of the circuit, and the interconnection block is configured to implement the connections between level i and $i+1$. We use the term *dynamic*, to describe the logic modules and the interconnection, because the function implemented by the logic modules and the interconnection structure is dynamically re-configured at every level. Note that programming of a circuit onto *Dharma* consists of a one-time compilation of the values to be loaded into the DLMs and DIAs. These values are then loaded onto the chip at power-on or 'boot time'.

A DLM generates an output which is a combinational function of its inputs. To allow the possibility of realizing any function, the DLM is an LUT (see Section 10.1 for other types of DLMs). All the DLMs can perform a look-up (based on the input configuration) in parallel. Hence, each level of logic of the input circuit can be evaluated in parallel by the DLM array (provided, of course, the number of functions being evaluated is less than the number of DLMs). The DIA connects the outputs of the DLMs and pass-buffers to the inputs of the next level of logic, through latches. To allow 100% routability, with predictable performance, the DIA block is designed such that all required connections are possible. The latches are used to store the signals between levels, and they also function as storage for state variables, in sequential circuit implementations. The pass-buffers are required to pass signals through levels; eg., if signal X is generated in level 2 and required in level 4, X will pass through level 3.

Based on the above four blocks, *Dharma* can be parameterized by the following variables (assigning different values to these variables will generate a chip-series).

1. Number of DLMs, C.

2. Number of inputs per logic DLM, K.

3. Number of pass-buffers, B.

4. Number of levels, L.

The number of internal latches would then be $KC + B$.

8.2.2 Detailed block diagram

Figure 8.6 is a detailed block diagram of the *Dharma* architecture.

In addition to the four main blocks illustrated in Figure 8.5, Figure 8.6 has the following:

1. *Level Generation Circuitry*, to count the level being implemented at the current clock instant.

2. *Internal Clocking Circuitry*, to generate the internal clock in synchronization with the external clock.

3. *Memory Write Circuitry*, to program the chip for a particular circuit.

The DIA block is split into two portions, an L-crossbar and a B-crossbar. The L-crossbar connects the outputs of the DLMs and the pass-buffers to the latches at the DLM inputs. The B-crossbar connects the pass-buffer outputs (after they are multiplexed with the primary inputs) to the latches at the pass-buffer inputs.

A DLM output which has to pass through to the next level is directly connected to a unique pass-buffer. This is a hard-wired connection. However, this connection is made to pass through a 2-to-1 multiplexer. The other input to the 2-to-1 multiplexer is from the B-crossbar. These 2-to-1 multiplexers are called *Buffer Muxes* and are shown in the lower right hand portion of Figure 8.6, to the left of the pass-buffer input latches. This interconnection scheme allows the pass-buffer to be used to pass other signals, when its corresponding DLM's output is not required at higher levels. In addition to the hard-wired connections, there are pass-buffers which are not associated with any DLM, and these are used to pass signals from pass-buffer outputs (the hard-wired connection is used to pass a signal the first time it is generated, i.e., by the DLM).

The chip's inputs (primary inputs) are latched into the *Primary Input Registers*. These latched signals enter the interconnection matrix through 2-to-1 multiplexers, called *PI Muxes*, shown atop the L-crossbar. The primary inputs are multiplexed with DLM outputs and pass-buffer outputs before entering the L-crossbar.

The outputs are latched into the *Primary Output Registers* (shown on the right side of Figure 8.6). The outputs can be from the DLMs directly, or can come through the

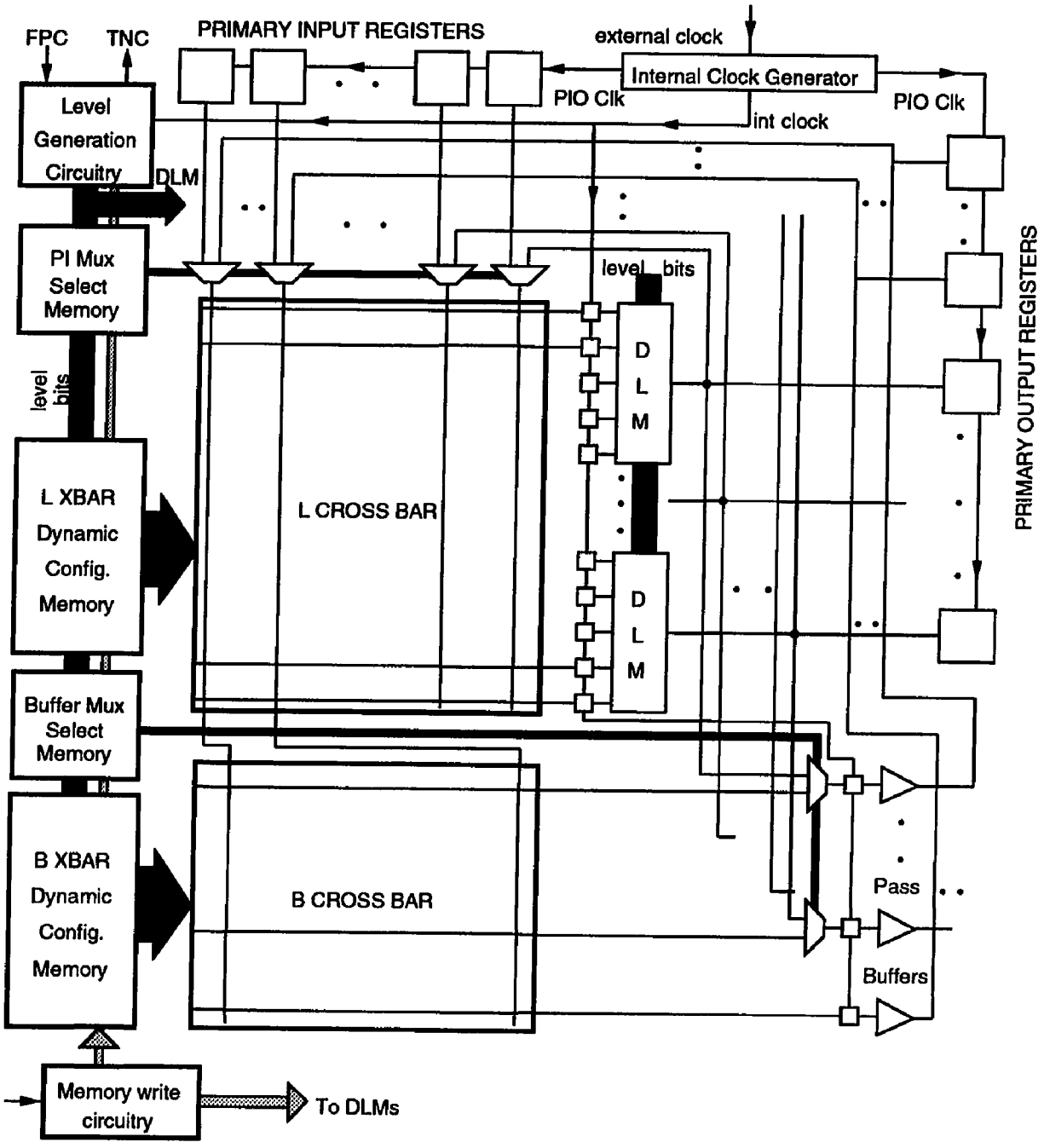


Figure 8.6: Detailed block diagram of *Dharma*

pass-buffers.

In Figure 8.6, the DLMs are 5-input, 1-output LUTs. Other kinds of DLMs are possible, and are explained in Chapter 10.

The *Level Generation Circuitry* keeps count of the level, and is controlled by the internal clock, the *FPC* signal, and the *Mode* bit (explained in Section 8.2.9). The current level is broadcast to the rest of the chip in the form of level bits.

The *L XBAR Dynamic Configuration Memory* block stores the configuration bits of the L-crossbar, and the *B XBAR Dynamic Configuration Memory* block stores the corresponding bits for the B-crossbar. These configuration bits are changed every level, and are addressed by the level bits.

The *PI Mux Select Memory* stores the select control for the *PI Muxes*. Each multiplexer requires 1 select bit, and this has to be changed at every level. Hence the *PI Mux Select Memory* is also addressed by the level bits. The *Buffer Mux Select Memory* block, likewise, stores the select control bits for the Buffer Muxes. These bits are also level dependent, and hence this block is also addressed by the level bits.

At power up, the *Memory write circuitry* loads in the values of the four memory blocks (PI Mux Select, L XBAR Dynamic Configuration, Buffer Mux Select and B XBAR Dynamic Configuration), the DLMs, and the level register in the *Level Generation Circuitry* (see Figure 8.11). These values are loaded in a serial manner from an external non-volatile memory (eg., an EPROM). The number of bits to be loaded into a chip is dependent on the on-chip resources (number of DLMs, number of pass-buffers, etc.). However, this number is invariant of the circuit being implemented on the chip, and hence can be preset at manufacture time. The memory write circuitry loads in the preset number of bits from external memory, and then stops. At completion, it signals the level generation circuitry to start the level count.

8.2.3 Clocking scheme

The *Internal Clock Generator* generates the internal clocks, labeled as *Int clock* and *PIO clock*. These clocks are synchronized with the external clock signal. The latches at the DLM inputs and the pass-buffer inputs are clocked by *Int clock*. *PIO clock* latches the chip inputs and outputs. There are two possible schemes of clocking.

1. EFP clocking scheme

2. S clocking schme

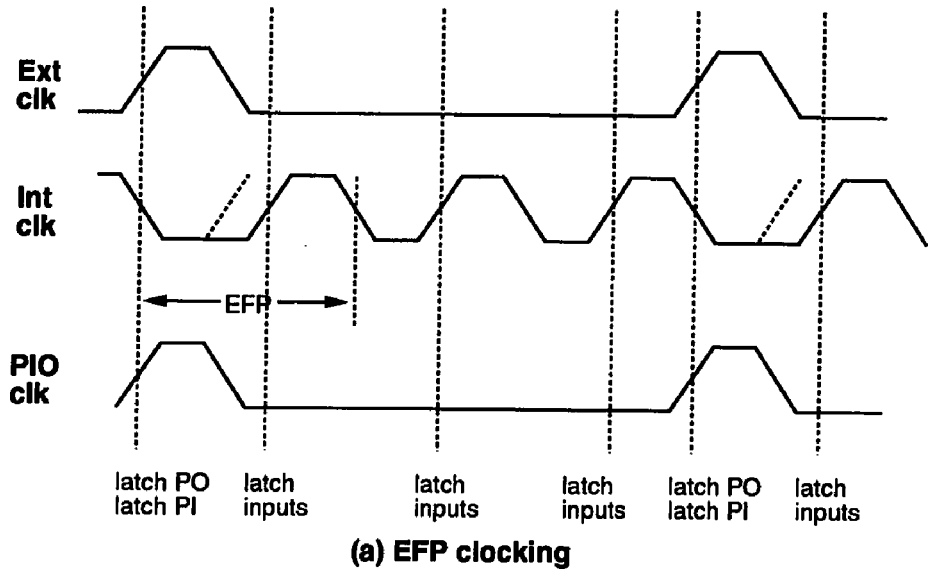
These clocking schemes are illustrated in Figure 8.7. The *Extended First Period* (EFP) scheme requires a non-uniform *Int clk*. The *Simple* or *Slow* (S) clocking scheme is simple, but results in a slower circuit. In the EFP scheme, the first period of the internal clock, following the rising edge of *Ext clk*, is longer compared to the rest of the periods. This first period is slowed down to accommodate the time required for the inputs to get latched into the input registers (see Section 12.2). Also, *Int clk* is 180° out of phase with *Ext clk*. *PIO clk* follows *Ext clk* exactly. However, the *Internal Clock Generator* must generate this explicitly (and not simply tie it to *Ext clk*) since there are applications (eg., a pure combinational circuit) which do not provide *Ext clk*. The S clocking scheme requires one additional *Int clk* cycle compared to the EFP scheme, for any circuit. Hence the "Slow" prefix. But, this scheme does not require the complex circuitry to generate extended first periods, and hence is very simple and straightforward. Here, *Int clk* is in phase with *Ext clk*.

8.2.4 Multi-chip operation

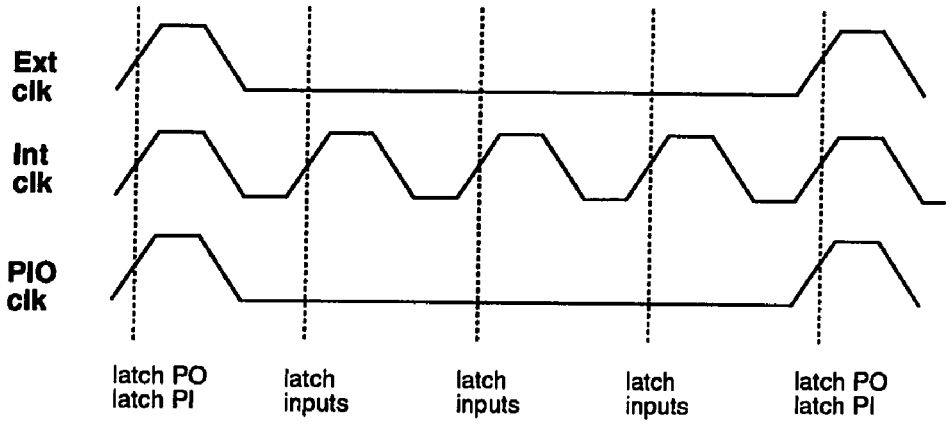
The *FPC* and *TNC* signals facilitate multi-chip operation. Such an operation may be desired when a circuit's levels exceed the maximum levels on a single chip, L. Let l be the number of levels in the circuit, and let $l > L$. If the number of DLMs in a level d, $\#(d)$, is such that $\#(d) < C, \forall d$, where C is the number of DLMs on a chip, then, although the circuit cannot be implemented on a single *Dharma* chip, its implementation can be spread across multiple *Dharma* chips. The number of such chips required would be $\lceil l/L \rceil$. These chips are connected in a chain-like fashion, with the first chip implementing levels $(0 \ldots (L-1))$, the second chip implementing levels $(L \ldots (2L-1))$, and so on. The outputs from the first chip will be connected as inputs to the second chip, the outputs from the second chip will be connected as inputs to the third chip, and so on, and finally the outputs from the last chip will be connected to the inputs of the first chip. Figure 8.8 shows this multi-chip interconnection scheme.

FPC stands for *From Previous Chip* and *TNC* stands for *To Next Chip* and it is obvious from Figure 8.8 why they are named thus.

The *Dharma* chips in Figure 8.8 have $L = 4$ each, and the circuit being implemented has $l = 10$. Hence $\lceil 10/4 \rceil = 3$ chips will be required, assuming that $\#(d) < C$,



(a) EFP clocking



(b) S clocking

Figure 8.7: Clocking schemes

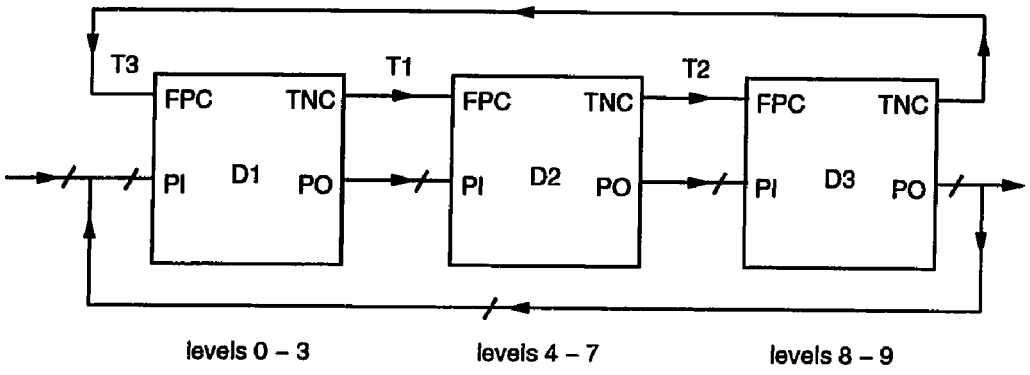


Figure 8.8: Multi-*Dharma* interconnection scheme. In this example, $l = 10$, and $L = 4$.

$\forall d$. Chip *D1* implements levels 0 to 3, chip *D2* implements levels 4 to 7 and chip *D3* implements levels 8 and 9. The waveform sketches for the *TNC* and *FPC* signals are shown in Figure 8.12 and further explanation can be found in the subsection on *Level Generation Circuitry*.

A *Mode* bit is provided on each chip to select between single chip and multi-chip operation. See the *Level Generation Circuitry* subsection (Section 8.2.9) for more details.

8.2.5 Dynamic logic modules

The logic module has to be general enough that it can be programmed to perform any combinational function of its inputs. Among the known logic modules are PLAs, look-up tables and MUX based modules. Among these, only the look-up table based modules can perform all the combinational functions of its inputs. Xilinx series of chips have used look-up table based logic modules, in various configurations, and also with more than one output per module. From the logic synthesis point of view, logic modules with just one look-up table with a single output, have been easier to handle, and mapping for delay, has been shown to be optimal, in this case [Cong 92]. In *Dharma*, we have chosen the logic module to be a K-input, 1-output LUT. However, it should be noted that the architecture is general enough to allow other types of logic modules (see Chapter 10). The advantages of using more complex configurations would depend on how well the synthesis tools handle such logic modules.

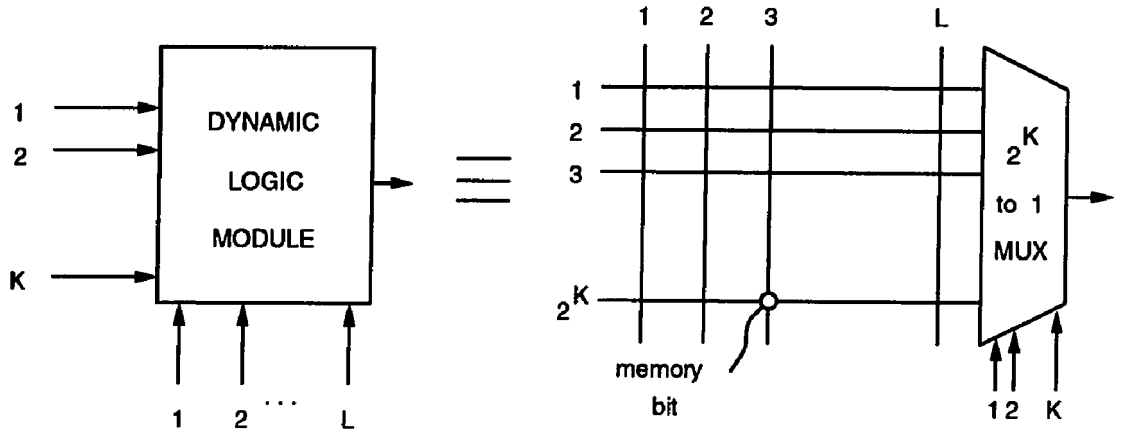


Figure 8.9: Dynamic logic module architecture

Each logic module may have to perform a different function at each level. A K-input, 1-output LUT requires 2^K bits of memory to implement any function of K-inputs. If a single chip can be used to implement circuits of at most L levels, then each logic module will require $L \times 2^K$ bits. A detailed diagram of the DLM is shown in Figure 8.9. There are many ways in which the level select lines and the K address lines can be combined together, and the figure shows one way. Which is the best one to use will depend on the area usage and memory read time, which would depend on the technology being used. Combining the level-selects and the K inputs together, is in essence making each DLM a $(K+log_2L)$-input, 1-output LUT, and this LUT should be designed such that area usage is kept down while giving timing performance as close to the K-input LUT as possible.

8.2.6 Pass-buffers

The pass-buffers are used to pass signals through levels, as already described in the previous section. The pass-buffer is only a pedagogical aid to deal with such signals. In the actual implementation, the pass-buffer can be replaced by a wire [1].

[1]The pass-buffer should not be confused with the line drivers required to drive signals at the inputs of the crossbars. Such line drivers will be needed in the actual implementation of the *Dharma* chip, and are not shown in Figure 8.6 or 8.5.

8.2.7 Dynamic interconnection array

Figure 8.10 is a cut-out of the interconnection related circuitry, from Figure 8.6. The breakup into two crossbars is done so as to reduce the number of crosspoints.

Let us analyze how this reduction occurs. To keep the analysis general enough, let us assume that each DLM has m outputs, and K inputs. If there are C DLMs and B buffers, then $(mC + B)$ output signals need to be connected to $KC + B$ input latches. If a single crossbar were used to perform this routing, there would be (KC+B)(mC+B) crosspoint, assuming a full crossbar. Instead, by using two crossbars (the L-crossbar and B-crossbar), as shown, the number of crosspoints is reduced to $(mC + B)KC + B^2$ (again, assuming full crossbars). Hence this reduces the number of crosspoints by mCB. If we assume that the number of buffers B is related to the number of DLMs C, by the relation $B = \beta C$, then the reduction in crosspoints, $\mathcal{R}_{cp}$ is

$$\begin{aligned}\mathcal{R}_{cp} &= \frac{mCB}{(KC+B)(mC+B)} \\ &= \frac{mC(\beta C)}{(KC+\beta C)(mC+\beta C)} \\ &= \frac{m\beta}{(K+\beta)(m+\beta)}\end{aligned}$$

For example, if $K = 5$, $m = 1$ and $\beta = 2$ (i.e., the number of buffers is twice the number of DLMs), then,

$$\begin{aligned}\mathcal{R}_{cp} &= \frac{2}{(5+2)(1+2)} \\ &= \frac{2}{21} \\ &= 9.5\%.\end{aligned}$$

If, instead $m = 2$, then,

$$\begin{aligned}\mathcal{R}_{cp} &= \frac{2 \times 2}{(5+2)(2+2)} \\ &= \frac{4}{28} \\ &= 14.3\%.\end{aligned}$$

The circuit inputs enter the interconnection matrix, multiplexed with the DLM and pass-buffer outputs. Input signals that are required at higher level in the circuit can be multiplexed with pass-buffer outputs (those pass-buffers that do not have a Buffer Mux at

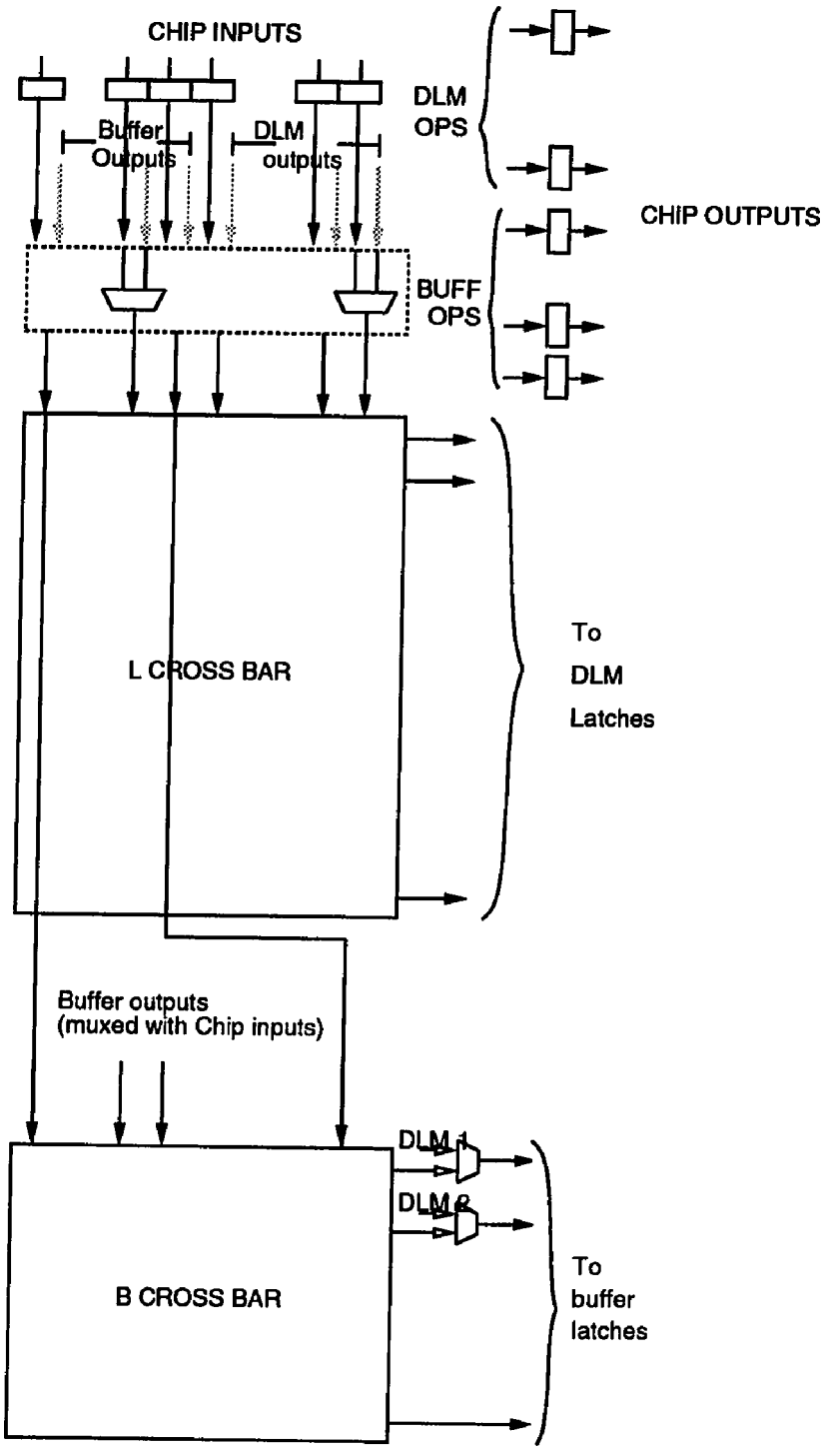


Figure 8.10: Dynamic interconnection block architecture

their input latch). In such cases, the input can be used directly (by setting the multiplexer select line to choose the input instead of the pass-buffer output) when that particular level is being evaluated. If the number of input signals required at higher levels exceeds the number of such pass-buffers, then some of the input signals will have to go through pass-buffers at lower levels.

There are many different ways in which to implement the crossbars, so as to reduce the number of switches and area occupied. Several such techniques are listed in Chapter 10. Whatever be the technique used, the idea is the same, to be able to connect the outputs of level i to the inputs of level $i+1$, so that all nets are routed.

8.2.8 Latches

The number of internal latches on the chip equals the number of inputs to the DLM and pass-buffers, i.e., $KC + B$. These latches serve 2 purposes:

1. To store the intermediate signals across levels
2. To store state variables of sequential circuits

The latches are clocked by the internal clock at a frequency of f_I. The latch value is unchanged during the level evaluation.

8.2.9 Level generation circuitry

An exploded view of the *Level Generation Circuitry* block of Figure 8.6 (upper left corner) is shown in Figure 8.11 [2]. It has the following components.

1. Number of circuit-levels register.
2. Level counter.
3. *Mode* bit, to choose between single chip and multi-chip operation.
4. Decision and control circuitry to control operation of the level counter and to generate the *TNC* signal.

[2]This diagram is slightly different from the one shown in [Bhat 93]. However, both perform the same function.

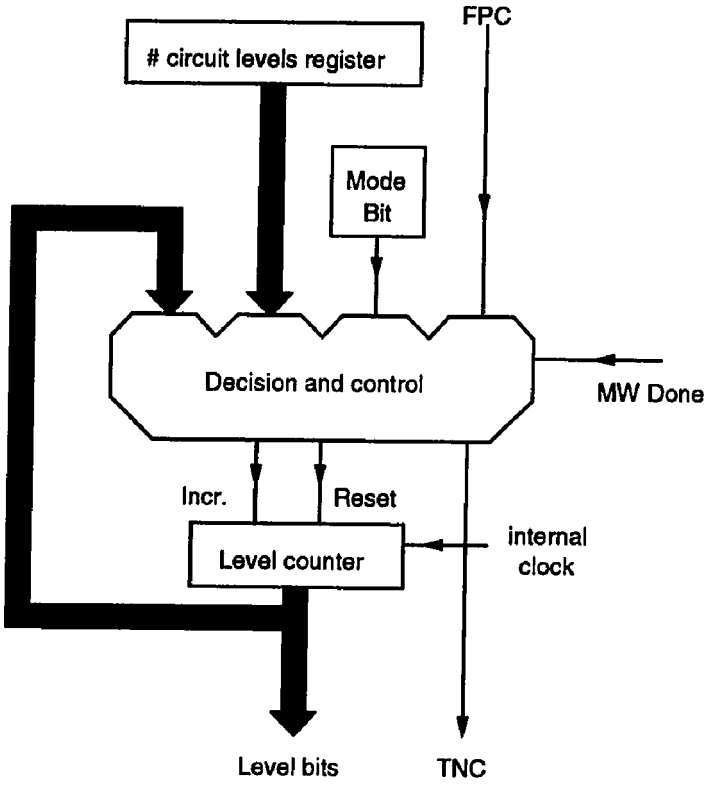


Figure 8.11: Block diagram of level generation circuitry.

The circuit-levels register stores the value of the number of levels the circuit has. This value must not exceed the maximum number of levels, L. The level counter's value is compared with this stored value and the counter is reset to zero when the two are equal. For example, if the circuit has 4 levels, then the level counter counts 0, 1, 2, 3, 0, 1, 2, 3, etc. This comparison operation is performed within the decision and control block. In Figure 8.8, chips *D1* and *D2* have the value 4 loaded in to their circuit-levels register, and chip *D3* has a value 2 in its circuit-levels register.

The level counter counts at every internal clock tick, when the *incr* signal is asserted. In single-chip operation, the *incr* signal is always asserted. In multi-chip operation, *incr* goes low when the count reaches a number which is one less than the value stored in the circuit-levels register.

The *Mode* bit selects between single chip and multi-chip operation. Although shown as a separate bit in Figure 8.11, it can be an external pin on the chip, which is pulled high or grounded, depending on how the chip must function.

Figure 8.12 shows the timing relationship amongst the *FPC* and *TNC* signals, for the example situation of Figure 8.8. The level counters in *D1* and *D2* count from 0 to 3, whereas the level counter in *D3* counts 0 and 1. The values of the level counters are shown symbolically by means of multi-level waveforms in Figure 8.12. *D1*'s level counter starts counting after its *FPC T3* goes high. *D1*'s *TNC* (same as *D2*'s *FPC*), *T1* goes high after it's level counter reaches 3. This starts *D2*'s level count at the next clock. *D1*'s level count then stops and waits for its *FPC* to go high. *T1* is lowered after one clock cycle. Similarly, *D2*'s *TNC* (same as *D3*'s *FPC*), *T2* goes high after *D2*'s level counter reaches 3, and *D3*'s level counter starts counting at the next clock. Finally, when *D3*'s count reaches 1, its *TNC* (i.e, *T3*) goes high and the cycle repeats.

The decision circuitry senses the *FPC* signal when the *Mode* bit is asserted (multi-chip operation). If the signal is high, then the level counter is reset, and *incr* is asserted. The *FPC* signal is then ignored. When the count reaches a number which is one less than the value stored in the circuit-levels register, the decision circuitry asserts the *TNC* signal, and de-asserts *incr*. *TNC* is de-asserted in the next clock cycle. The decision circuitry again senses the *FPC* signal. *incr* remains de-asserted till *FPC* is asserted, after which the cycle repeats.

The *MW Done* (memory write done) signal is from the *Memory Write* block. On power up, the level counter is reset. The decision circuitry then waits for *MW Done*. When

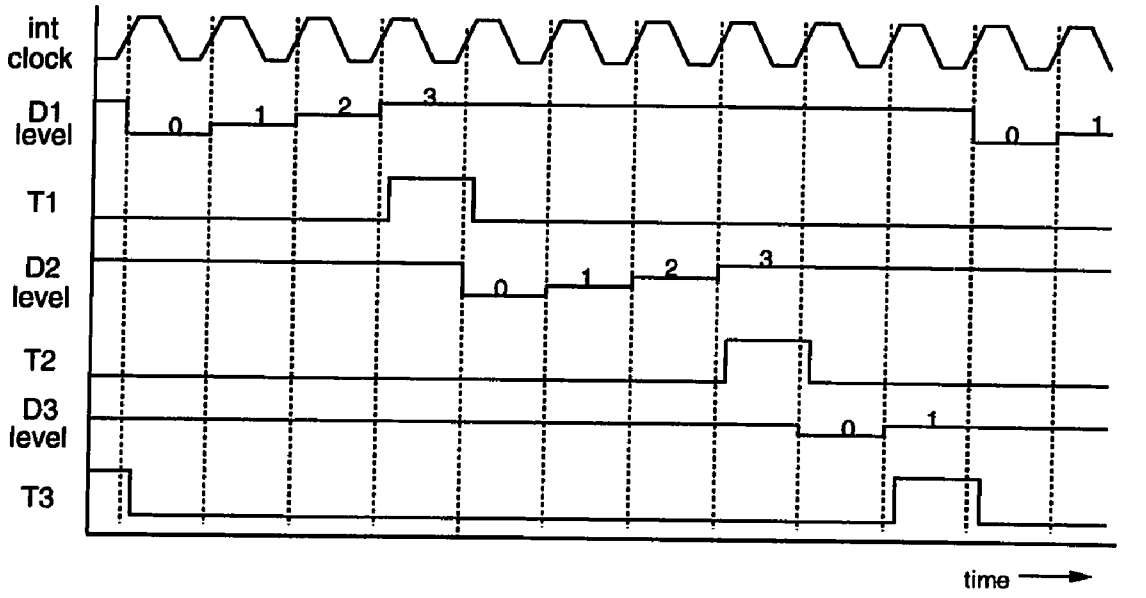


Figure 8.12: Timing waveforms for multi-chip operation.

this signal is asserted, *incr* is asserted, and the level counter starts counting from 0.

8.3 Operating procedure

To use *Dharma* to implement a given circuit design, the user must first convert the design specification into *Dharma*'s logic blocks, using a design flow similar to Figure 1.4, using software tools (see Chapter 11). The *Device Programming* software generates the bits that go into the various memory blocks and DLMs (see Figure 8.6). These bits are then stored in a non-volatile memory, such as an EPROM.

On power-up, the memory write circuitry of Figure 8.6 addresses this external memory (external to the *Dharma* chip, that is), and reads in the stored values therein. The bits are then loaded into the various memory blocks. This loading procedure will have to be repeated every time the power is switched on, since the memory bits on the *Dharma* are volatile [3]. After all the bits have been loaded (a fixed number for a given chip), the memory

[3]For this discussion, we have assumed that *Dharma* uses an SRAM technology. Depending on the DIA and DLM architecture, it is possible to have a non-volatile memory on the *Dharma* chip, in which case the loading procedure would be replaced by a one-time programming procedure to program the non-volatile

write circuitry's task is complete, and it remains idle for the rest of the time.

After the memory configuration bits are loaded, *Dharma* implements the circuit in a level by level fashion. Figure 8.13 illustrates the mechanism for a circuit with 3 levels. The figure shows the activities starting from power-up. In the figure, we assume an EFP clocking scheme (see Figure 8.7). After the memory bits are loaded, the *MW Done* signal is asserted. This signals the *Level Generation Circuitry* to start the level counter. The DIA is first configured to connect the Primary Inputs to the first level of logic (i.e., signals from Level 0 to Level 1). In the figure, this is shown on the row labeled *DIA Recnfg* as 'L0-L1', and we assume that this configuration takes half the internal clock cycle. At the negative edge of *Int clock*, the DIA is ready. The Primary inputs are latched at this instant . In the figure, this time instant is labeled *Latch PI* [4].

During the negative half cycle, the primary input signals propagate through the DIA. At the next positive going edge, the signals are latched at the DLM and pass-buffer inputs (marked *Latch inputs* in the figure). As the DLMs evaluate the functions, the DIA is reconfigured to connect the next level, L1-L2. The signal propagation through the DLM and the reconfiguration of the DIA occur in parallel. The DLM is assumed to be a LUT, hence it does not have a reconfiguration time. However, if it were a different type of DLM which needed reconfiguration, such a reconfiguration could be done in parallel with the DLM propagation period. The operation continues in similar fashion, till power is turned off.

After the DLM has evaluated the uppermost level of logic (Level 3), the output signals can be latched into the Primary output registers, and the new primary input signals can be latched into the Primary input registers. The dynamic operation is transparent to the external world. Note that the rate at which the PI and PO registers can be latched (i.e., the rate of the external clock), is $\frac{1}{(1+\delta)}$ times slower than the rate of *Int clock*, as explained in Section 12.2, equation 12.8.

The time instants marked ϵ are referred in Chapter 9.

The DIA and DLM propagation times times need not be equal. The duty cycle of the internal clock is set to the ratio $\frac{\max(t_D,t_R)}{t_I}$, where t_D is the sum of the DLM and latch propagation times, t_R is the sum of DIA reconfiguration time and latch hold time, and t_I is the sum of DIA propagation time and latch set-up time (see Section 12.2). A duty cycle

memory bits.

[4] In the label *Latch PI* and *Latch inputs*, the word *Latch* is used as a verb, and not as a noun.

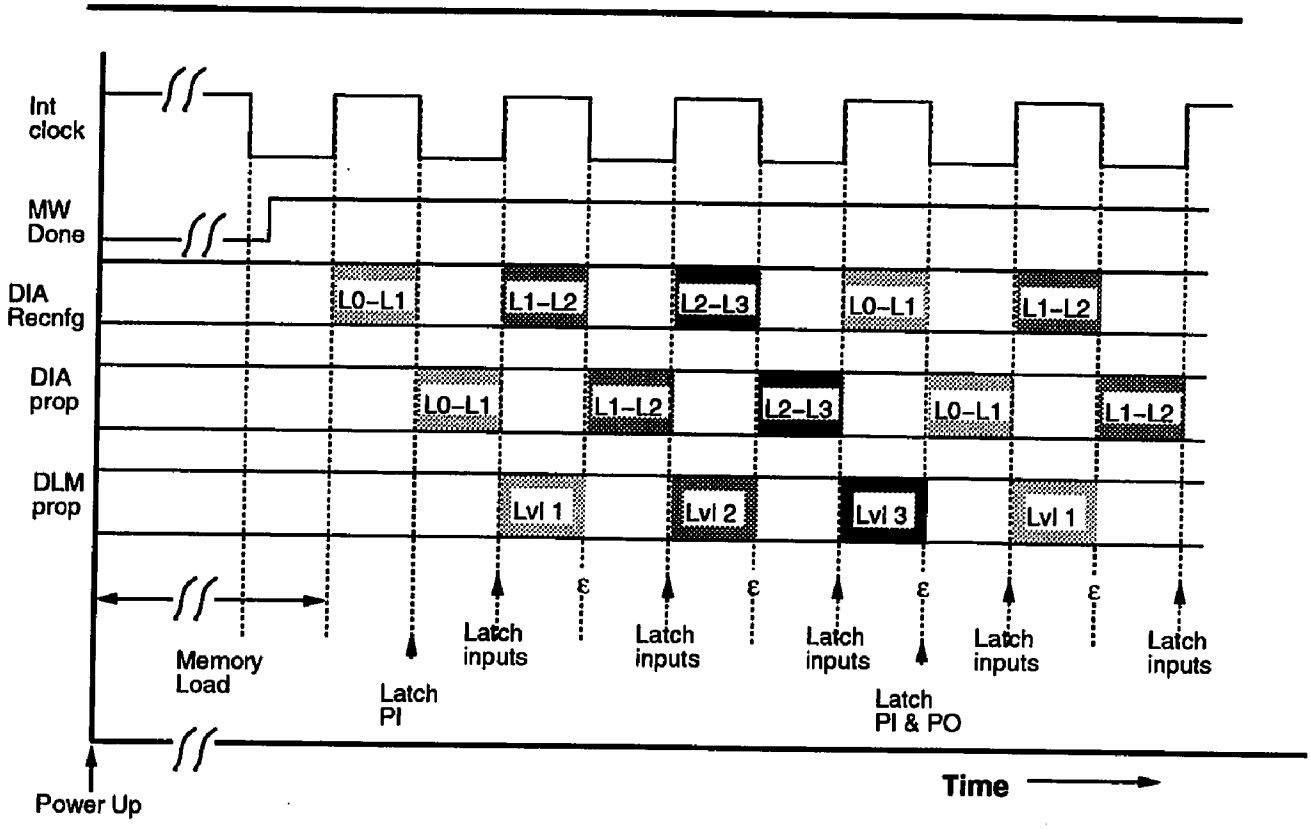


Figure 8.13: Timing diagram to illustrate *Dharma*'s operation. The circuit design has 3 levels.

other than 50% does not change the activities described above in any way.

From Figure 8.13, we see that there is no necessity for any global control logic, to supervise the chip operations. All operations are synchronized with respect to the internal clock and the level bits.

Chapter 9

Examples to Illustrate Dharma

In this chapter, we illustrate *Dharma* by means of simple examples. We have chosen two combinational and one sequential example, with a small number of logic elements, and we demonstrate by means of figures, how the circuits are realized on a *Dharma* device. For the purposes of the illustration, we have chosen a simplified version of *Dharma*. This simplified architecture, called *Simple Dharma*, is discussed first. *Simple Dharma* retains the main concept of Figure 8.5, viz., levelizing and time sharing of resources. However, the number of DLMs and the size of the DIA is kept small, so as to help understand the concepts better. The first example is very simple, and is more of a warm up exercise. The second example is more involved. The third example demonstrates that sequential circuit realization is very much like the combinational one.

9.1 *Simple Dharma*

Dharma consists of DLMs, DIA, pass-buffers and latches. The number of inputs and outputs per DLM, the number of DLMs, the number of pass-buffers, the interconnection structure of the DIA, the number of levels that can be implemented, etc., can all be set to different values so as to yield a family of *Dharma* chips. For this simple illustration, we use a hypothetical, simple architecture. This architecture has the following features:

- All DLMs are identical.
- Each DLM has 3 inputs ($K = 3$) and 2 outputs.
- There are 3 DLMs ($C = 3$).

- There are 5 pass-buffers ($B = 5$).
- The B-crossbar and L-crossbar are implemented as full crossbars.
- There are 5 chip inputs and 6 chip outputs.
- $L = 5$, levels of logic can be implemented (i.e., each crossbar point has 5 bits of memory to choose the configuration, and each DLM has $2^3 \times 2 \times 5 = 80$ bits to store the logic functions).

Figure 9.1 is a block diagram of this simple architecture. The crosspoints in the L-crossbar and B-crossbar are shown as shaded circles. The shaded circles represent the switch at the crosspoints and the memory bits required to configure the switch (to be either on or off). The chip inputs enter the interconnection array via multiplexers. In the figure, the inputs are shown to be multiplexed only with the pass-buffer outputs. As in Figure 8.6, inputs can also be multiplexed with DLM outputs, but for the purposes of this simple architecture, those multiplexers are unnecessary. Each multiplexer is individually selected. The multiplexer selection memory has 5 bits per multiplexer. The DLM outputs are connected to the pass-buffer inputs by means of multiplexers. Each of these multiplexers also has a separate select signal, controlled by 5 memory bits per multiplexer. To keep the diagram simple, the multiplexer selection memory is not shown. The DLM and pass-buffer inputs are latched; and these latches are clocked by an internal clock.

9.2 Combinational example 1

This is a very simple example, and is meant more as an introduction. Figure 9.2 shows the schematic diagram for this example circuit. There are 5 inputs, a, b, c, d and e, and 2 outputs x and y.

In terms of boolean equations,

$$\begin{aligned} G &= a + b + c \\ H &= d\,e \\ x &= G\,H' + G'\,H \\ y &= c + H \end{aligned}$$

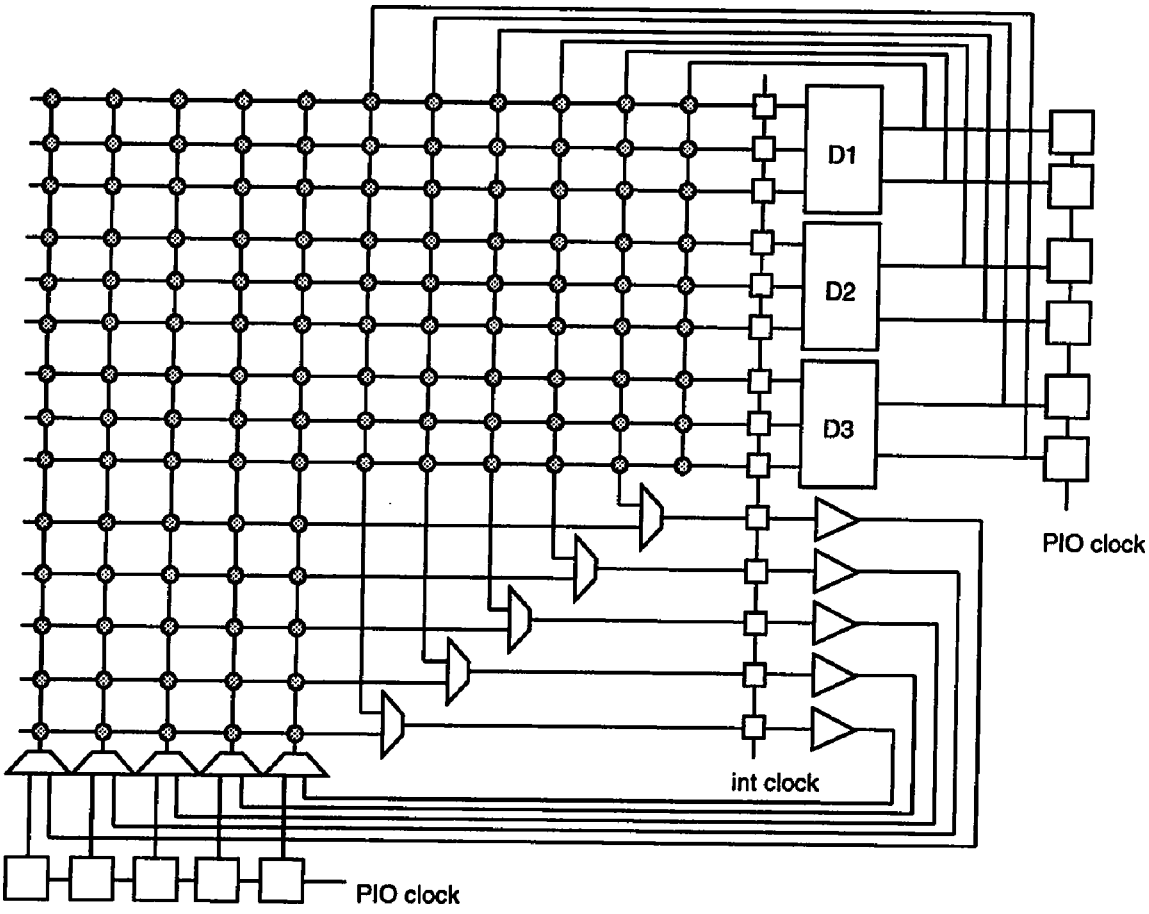


Figure 9.1: *Simple Dharma*

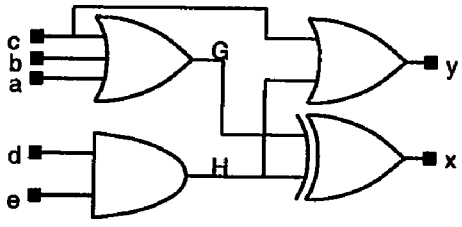


Figure 9.2: Schematic diagram for Example 1

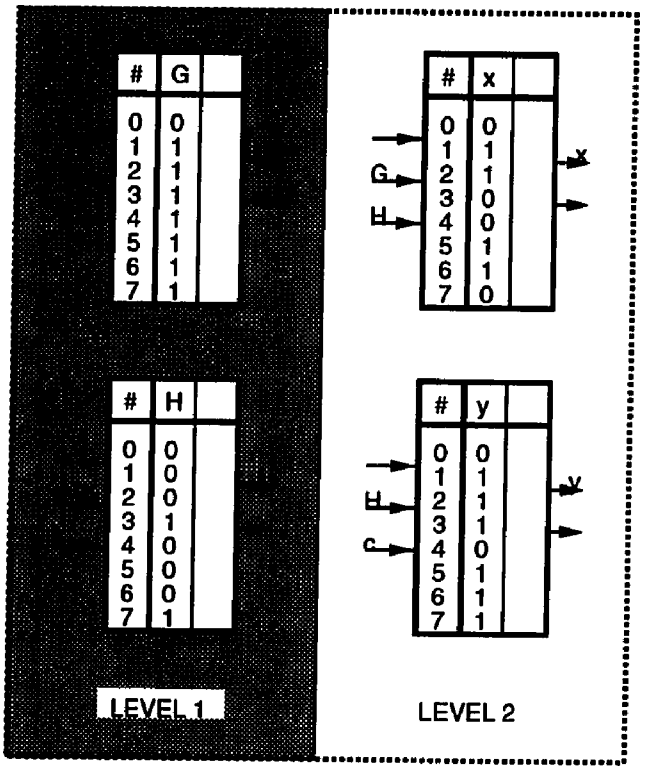


Figure 9.3: DLM values for Example 1

From Figure 9.2, we see that the circuit has 2 levels of logic. The realization of this example on *Simple Dharma* is explained in the following figures. Figure 9.3 shows the memory bits stored in the DLMs. In the first level inputs a, b, c, d and e are connected to the DLMs, and outputs G and H are generated. In the second level, G, H and c are connected to the DLMs to generate the outputs x and y.

Consider the bits stored to generate function G. G is the OR of its inputs a, b and c. Hence it takes the value 1 whenever any input is 1, and takes the value 0 only when all the inputs are 0. The memory addressed when a, b and c are all 0, is 0, and hence a 0 is stored at location 0. All the other memory locations have a 1 stored in them.

Next consider function H. H is the AND of its inputs d and e. Hence it takes the value 1 only when both inputs are 1. Since the DLM is a 3-input LUT, and H has only 2 inputs, half the memory locations are a duplicated. Signals d and e are connected to the

least significant bits of the memory address, and the most significant bit is not needed (an unused input can be connected to either Vcc or ground). d and e, both being 1 corresponds to memory addresses 3 (011) and 7 (111). Hence 1s are stored in these two locations. The rest of the memory bits are 0s.

The functions x and y are similarly programmed.

Figures 9.4 and 9.5 show the interconnection structure corresponding to Example 1. The signal values marked in these figures correspond to the time instants marked ϵ in Figure 8.13. That is, the DLM has just generated signals at a particular level, and the DIA is configured to connect the next level. A black crosspoint represents a connection between the horizontal and vertical lines. A white crosspoint implies that the crossing lines are not connected. Multiplexer selection is shown by means of a line through the multiplexer symbol. This line connects the selected multiplexer input to the multiplexer output.

9.3 Combinational example 2

Figure 9.6 shows the schematic for Example 2. There are 5 inputs, a, b, c, d and e, and 3 outputs, x, y and z.

In terms of boolean equations,

$$
\begin{aligned}
G &= a\,e \\
H &= c\,d' + c'\,d \\
I &= b\,d \\
F &= a\,e' + a'\,e \\
J &= I\,H'\,F \\
K &= b\,c + b\,d + c\,d \\
x &= G\,K + J \\
P &= c\,d \\
y &= H'\,F'\,b + H'\,F\,b' + H\,F'\,b' + H\,F\,b \\
M &= F\,H\,b' + F\,H'\,b \\
N &= K + G + M \\
O &= b\,c'\,d + b\,c\,d'
\end{aligned}
$$

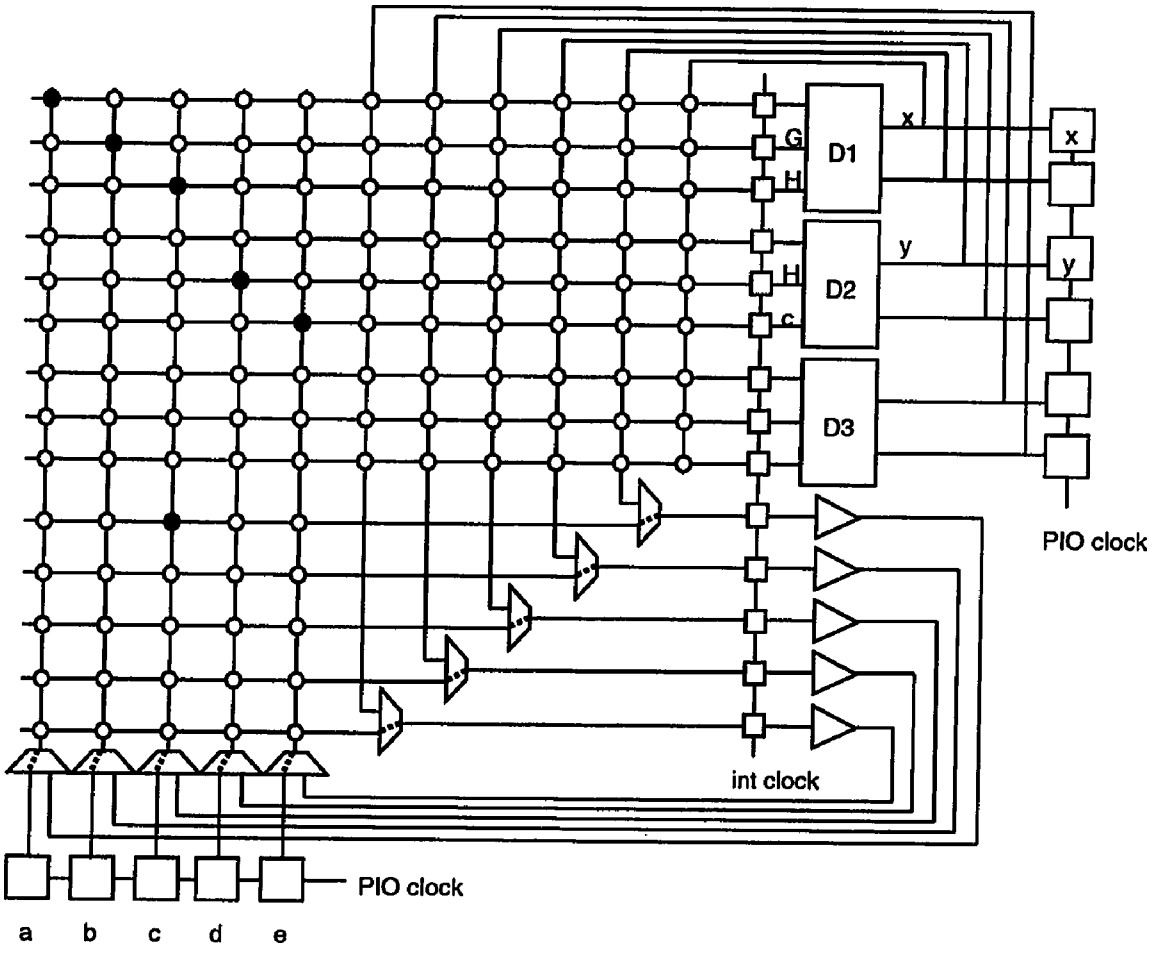


Figure 9.4: Example 1. DIA: Primary inputs → level 1; DLMs : Level 2.

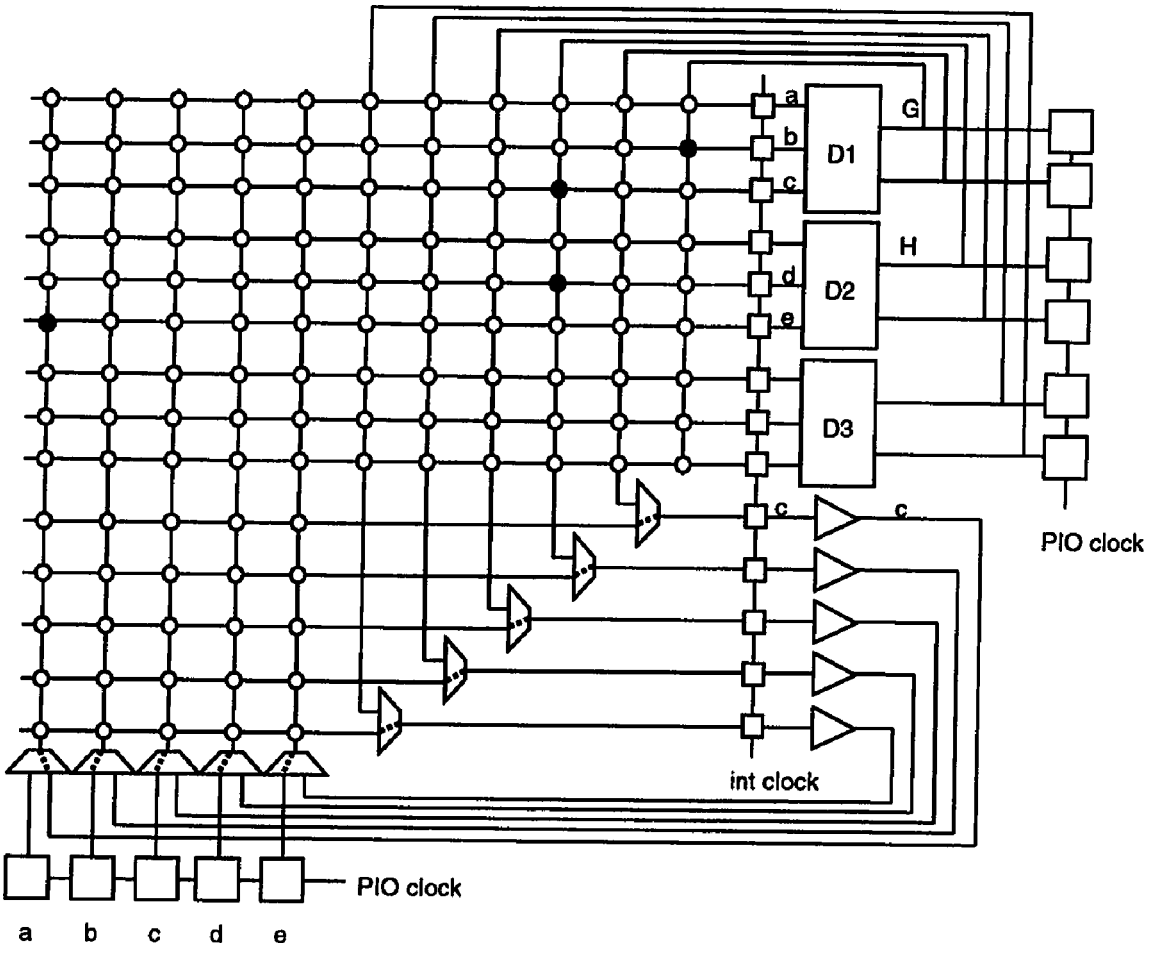


Figure 9.5: Example 1. DIA: Level 1 → Level 2; DLMs : Level 1.

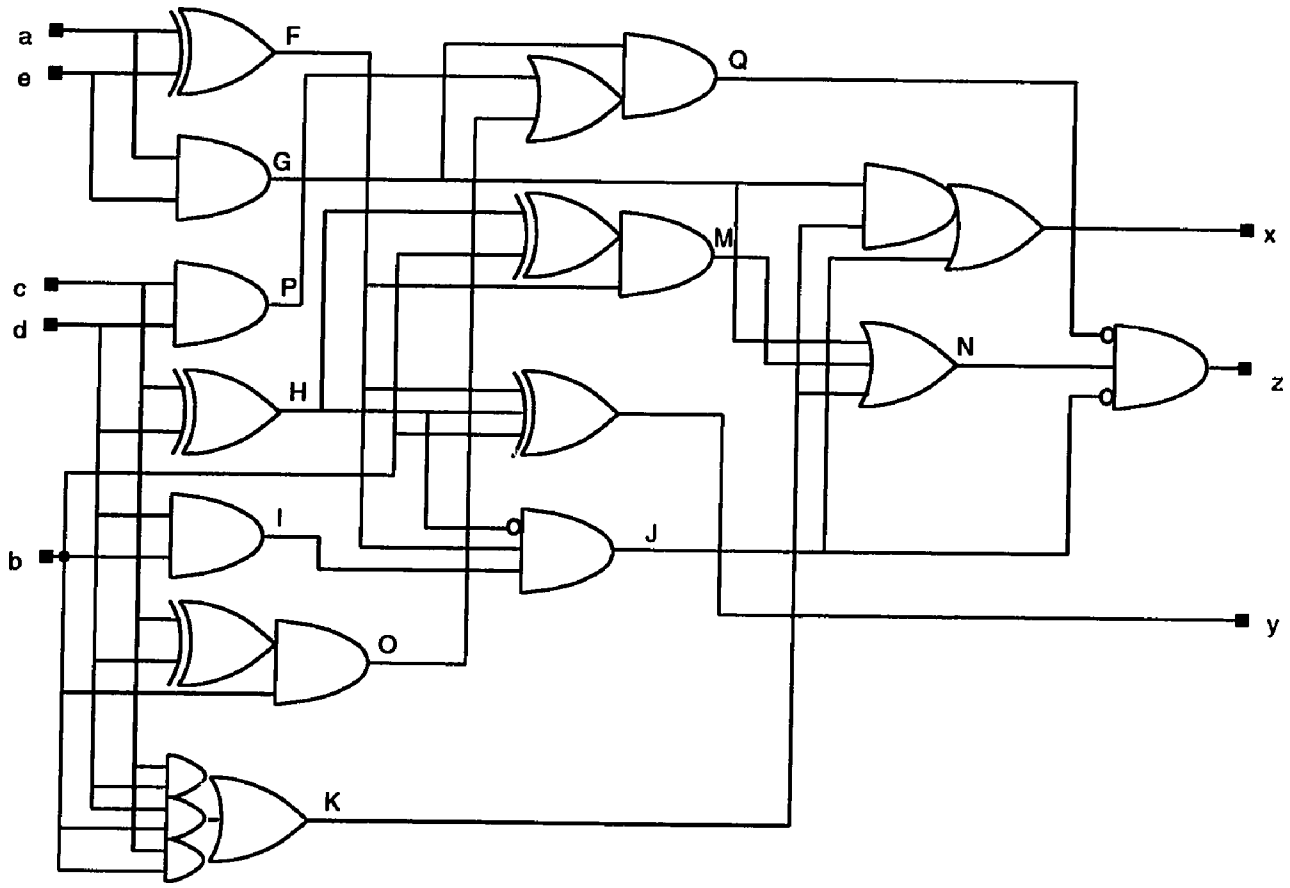


Figure 9.6: Schematic diagram for Example 2

$$Q = G\,P + G\,O$$
$$z = N\,J'\,Q'$$

From Figure 9.6, we see that the circuit has 4 levels of logic. Note that topological levelization (schematic Figure 9.6 has been drawn topologically levelized – gates having the same topological level (see Chapter 11), from primary inputs, are vertically aligned) cannot be used to realize this circuit directly on *Simple Dharma*, since level 1 has 7 gates, and there are only 6 DLM outputs per level. In Figure 9.7, we show the level assignments to the various functions, after doing a temporal partitioning (Chapter 11). Figure 9.7 also shows the DLM memory bits required to perform the assigned functions.

In Figure 9.7, we note that x and y have been assigned to level 4 (topological level of x was 3, and that of y was 2). Also, K, I and G have been moved to level 2, and Q and J have been moved to level 3. Functions that share common inputs have been assigned to the same DLM (eg., H and O; and K and I).

Figures 9.8, 9.9, 9.10 and 9.11 show the interconnection configuration for the various levels. As in Example 1, the signal values correspond to the time instants marked ϵ in Figure 8.13.

9.4 Sequential example

Figure 9.12(a) shows the circuit schematic of a simple sequential circuit (this is a bus arbitration circuit, and is a modified version of an example in [Alford 89]). The inputs are *R1* and *R2* and outputs are *O1* and *O2*. Signals *S1* and *S2* are the state variables. The (b) part of the figure shows only the combinational part of the finite state machine shown in (a). *PS0* and *PS1* are the *previous* (hence the prefix 'P') of the state variables *S0* and *S1* respectively. The circuit has a two-phase clock. This example therefore also serves to illustrate how multi-phase clocked circuits can be mapped to *Dharma*.

Figure 9.12(c) shows the phase relationship between the two clocks, $\phi 1$ and $\phi 2$. The positive edge of $\phi 2$ must lag the corresponding edge of $\phi 1$ by $\delta 1$. $\delta 1$ corresponds to the time required to generate signals *S1* and *S2*. Similarly, the next positive edge of $\phi 1$ should arrive only after a time delay of $\delta 2$, where $\delta 2$ includes the latch propagation time and the propagation through the rightmost combinational gates in Figure 9.12(b). The input and output registers are clocked by the positive edge of $\phi 1$ and the state registers are clocked

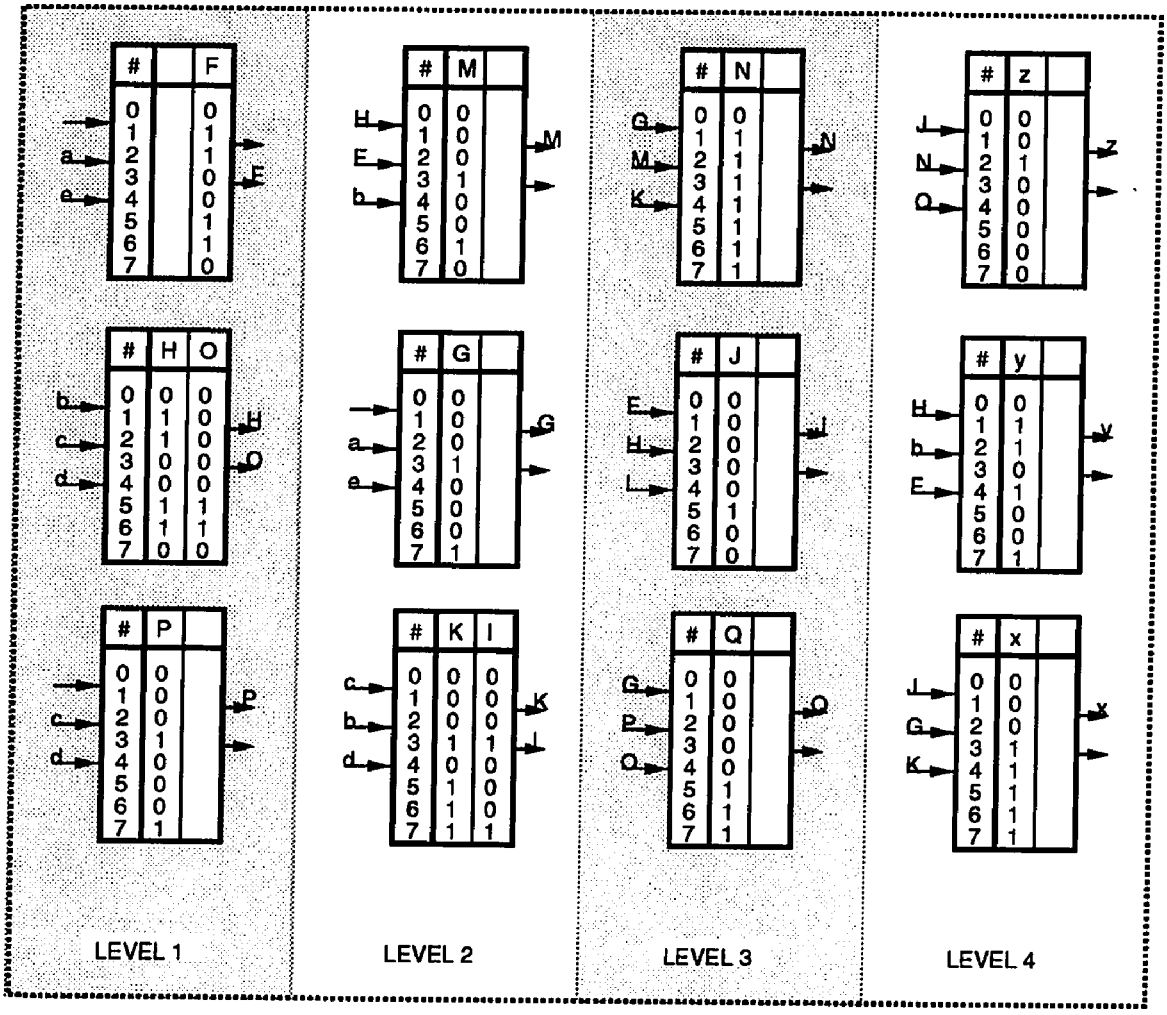


Figure 9.7: DLM values for Example 2

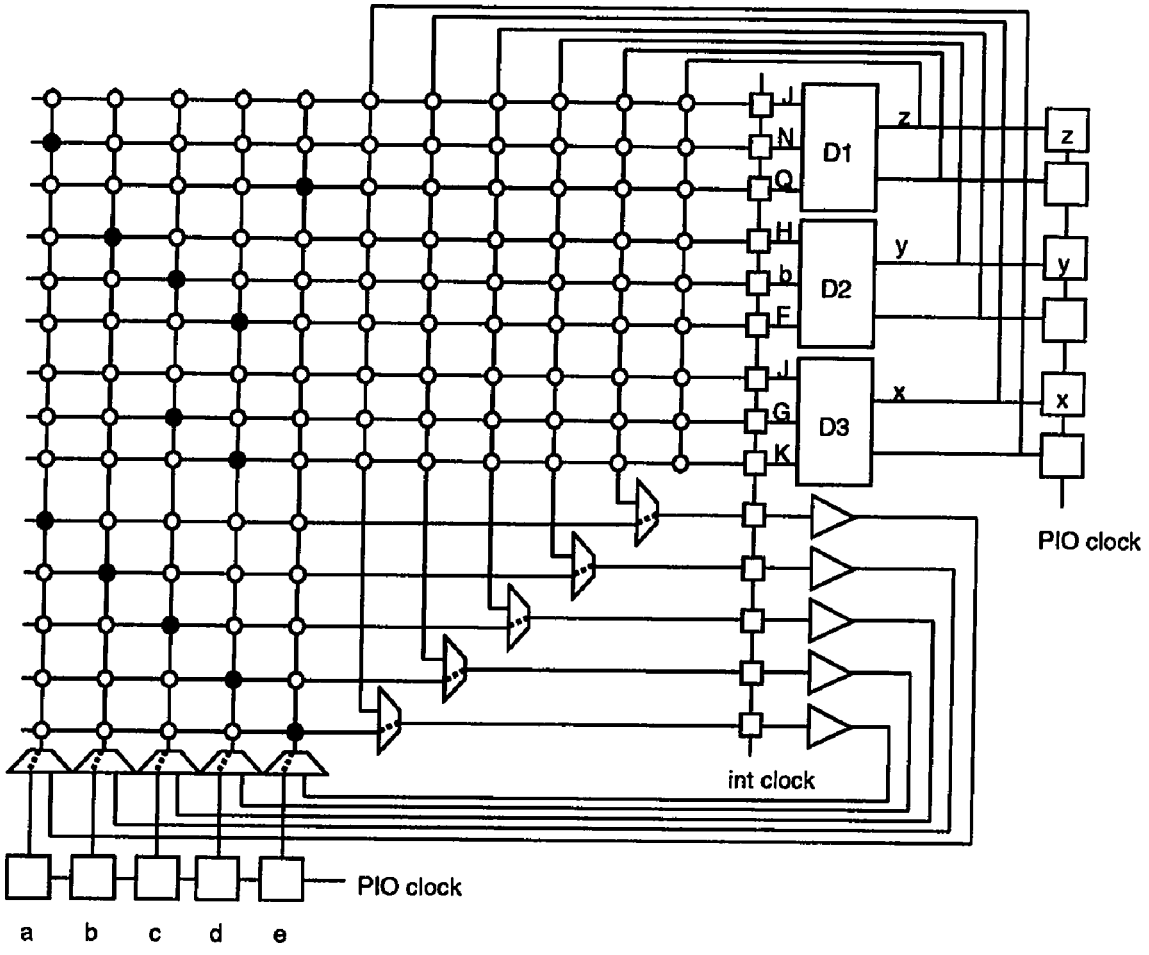


Figure 9.8: Example 2. DIA: Primary inputs → Level 1; DLMs : Level 4.

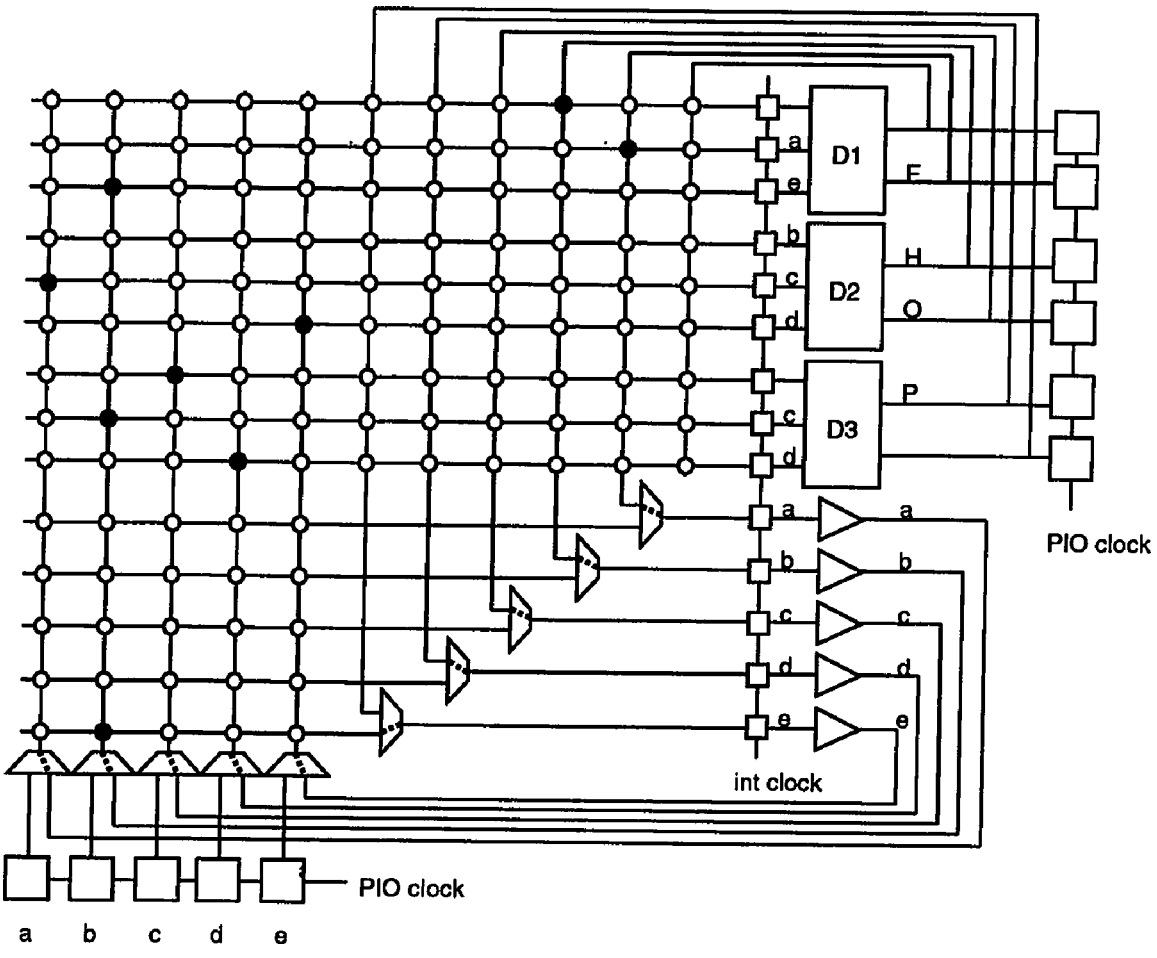


Figure 9.9: Example 2. DIA: Level 1 → Level 2; DLMs : Level 1.

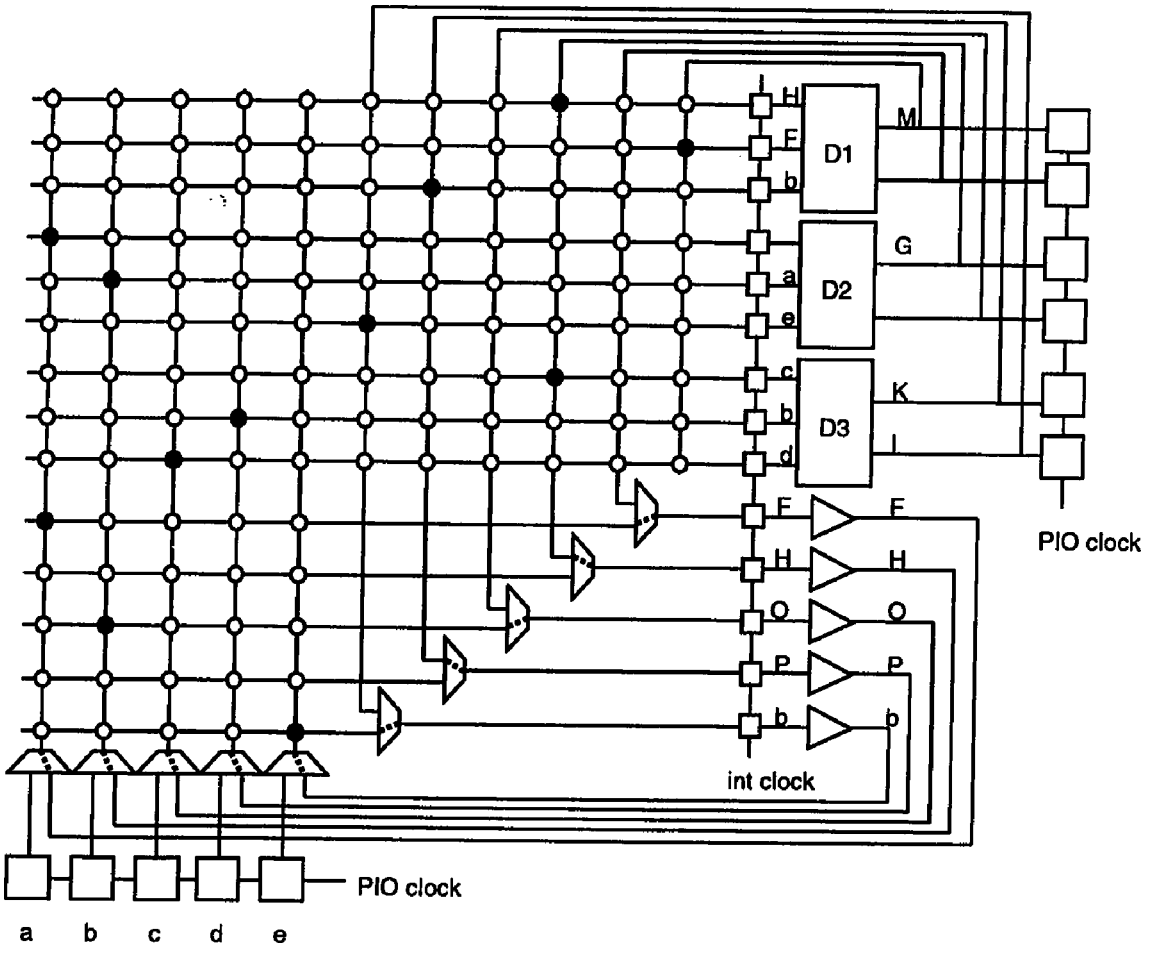


Figure 9.10: Example 2. DIA: Level 2 → Level 3; DLMs: Level 2.

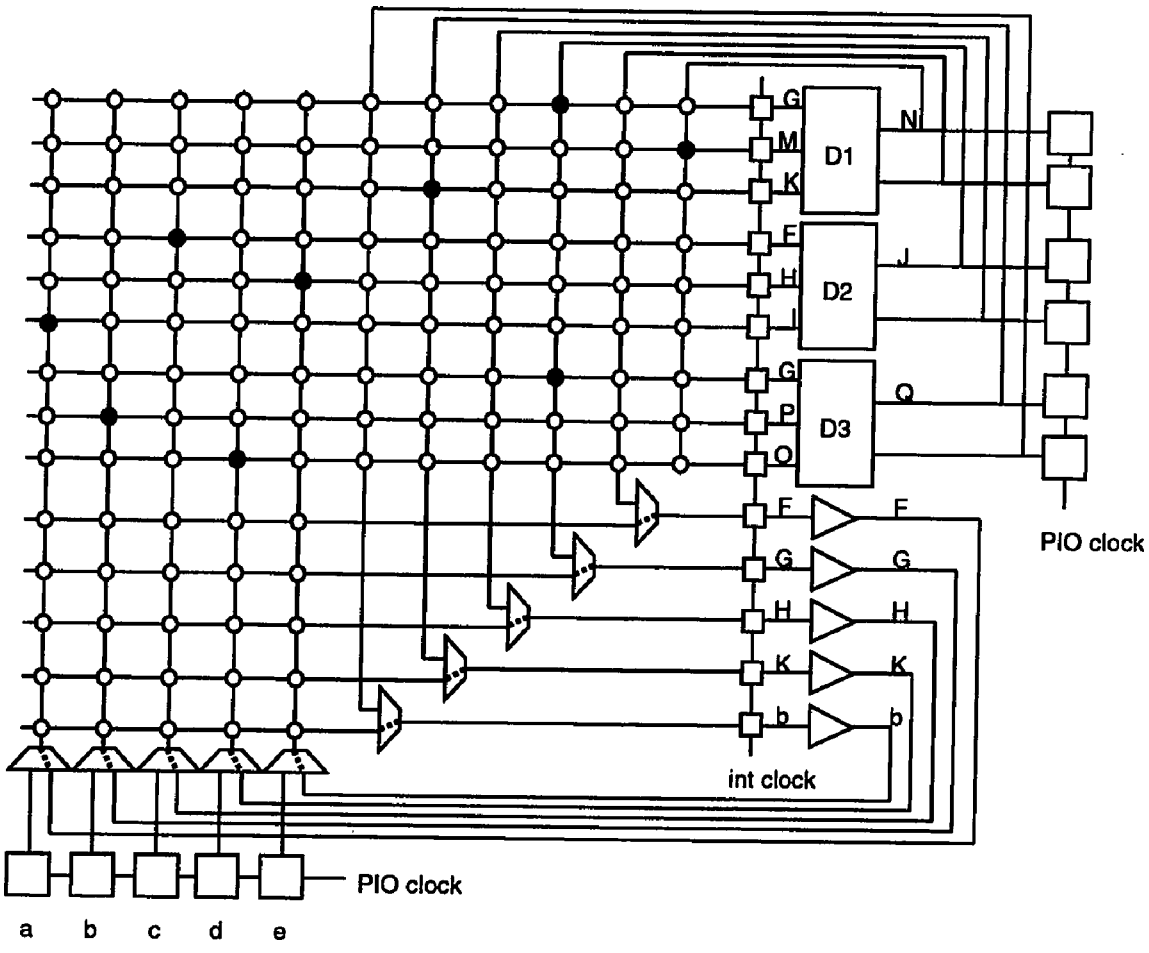
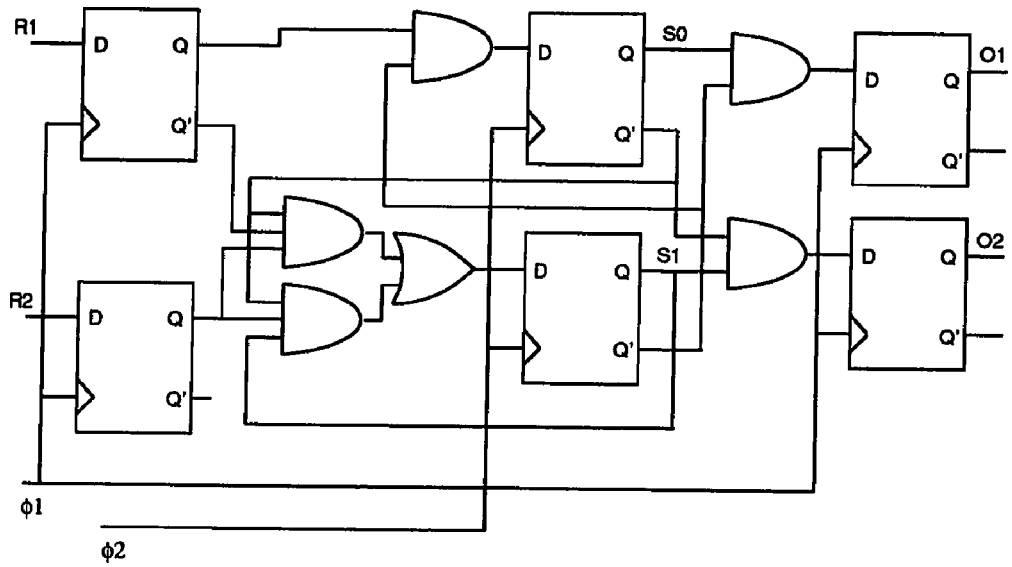
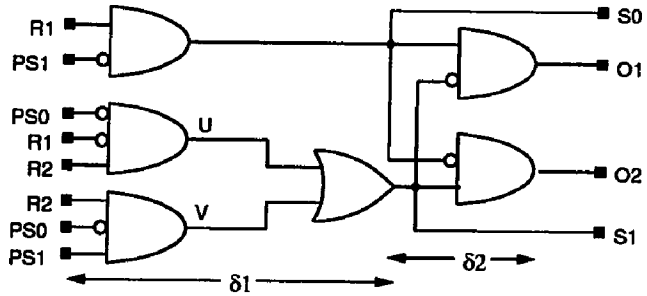


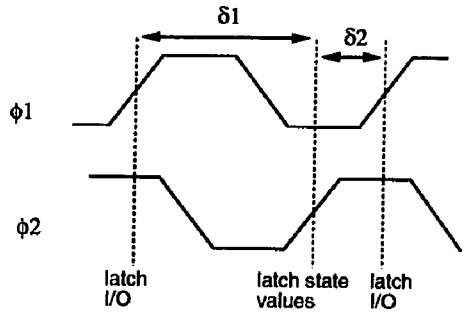
Figure 9.11: Example 2. DIA: Level 3 → Level 4; DLMs: Level 3.



(a) Circuit schematic



(b) Combinational part



(c) Clocking scheme

Figure 9.12: Sequential Example

by the positive edge of $\phi 2$.

Figure 9.13 shows the LUT values corresponding to Figure 9.12(b). The *Dharma* realization has 3 levels and requires only *one* phase of the clock, $\phi 1$. This clock can run at approximately one-third (see Section 12.2) the speed of the internal clock. Figures 9.14, 9.15 and 9.16 show the DIA configuration bits, and DLM signal values corresponding to time instants marked ϵ in Figure 8.13. These are very similar to the combinational examples, except that the state variables are fed back into the DIA, at the last level, and become the *PSi* ($i = 0$ or 1) signals.

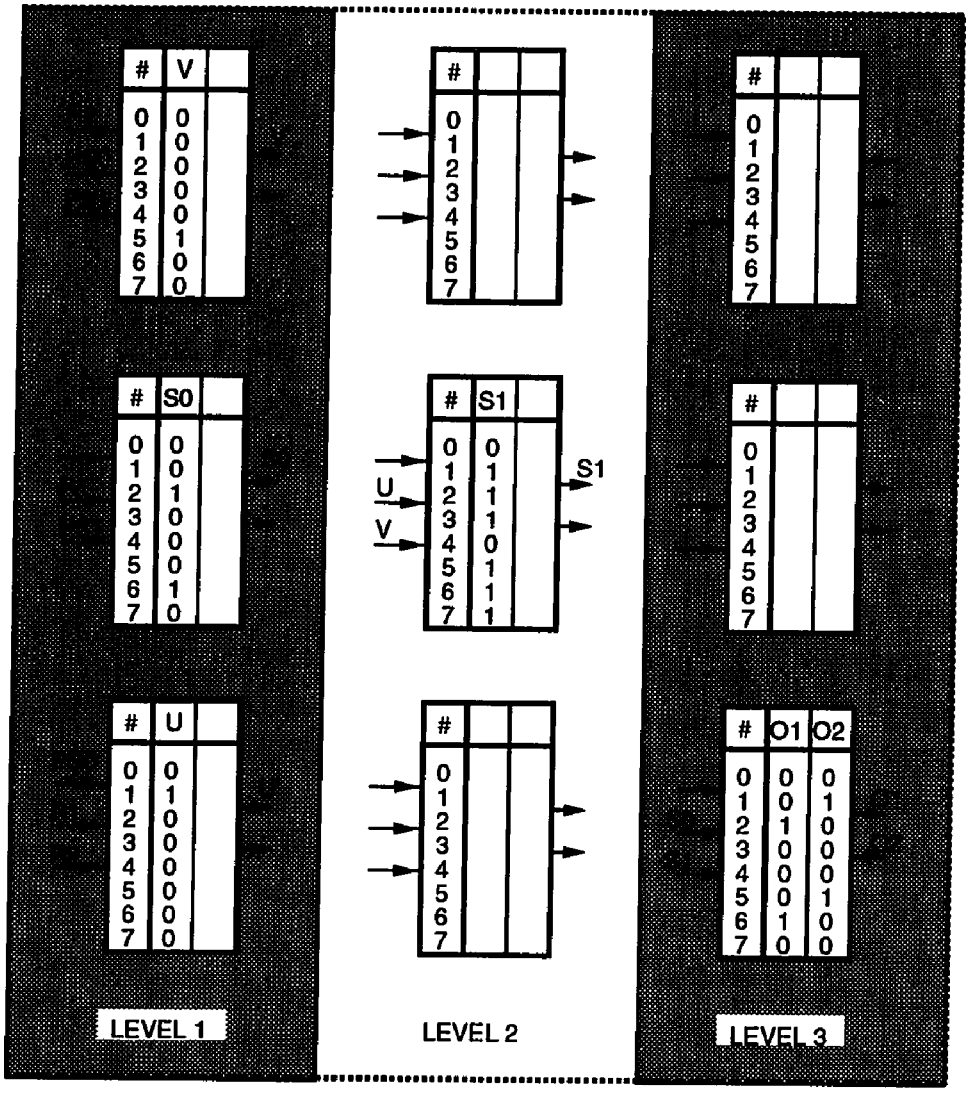


Figure 9.13: DLM values for the sequential example

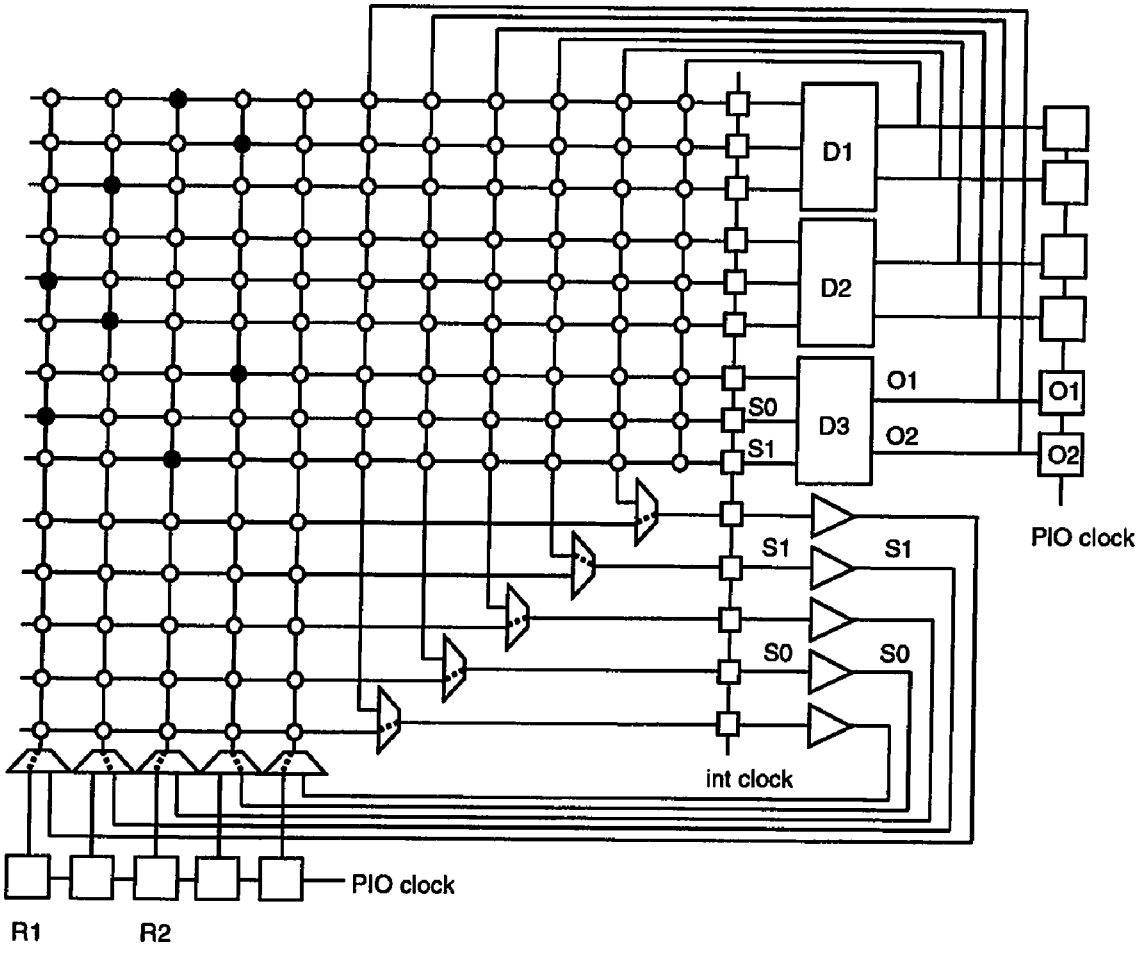


Figure 9.14: Sequential Example. DIA: Primary inputs, state variables → Level 1; DLMs: Level 3.

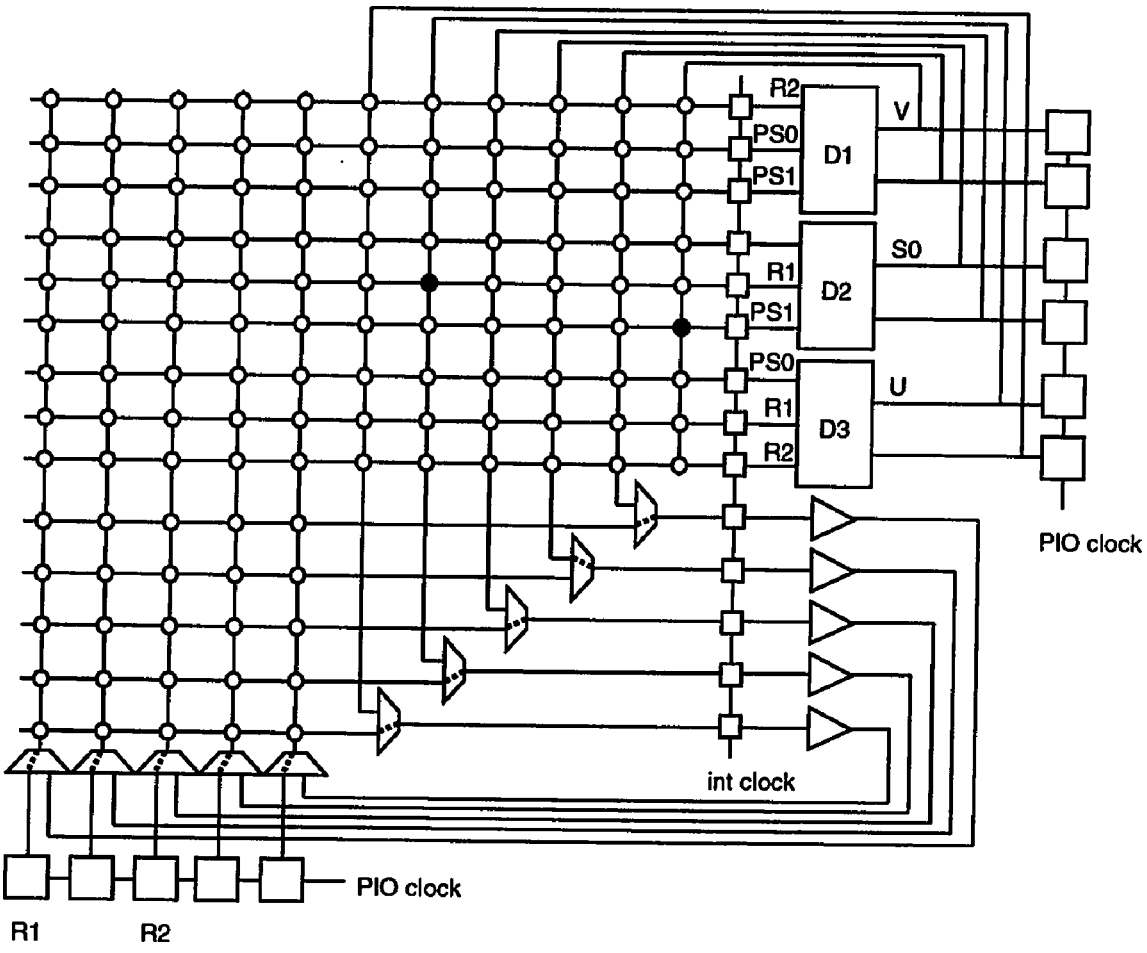


Figure 9.15: Sequential Example. DIA: Level 1 → Level 2; DLMs: Level 1.

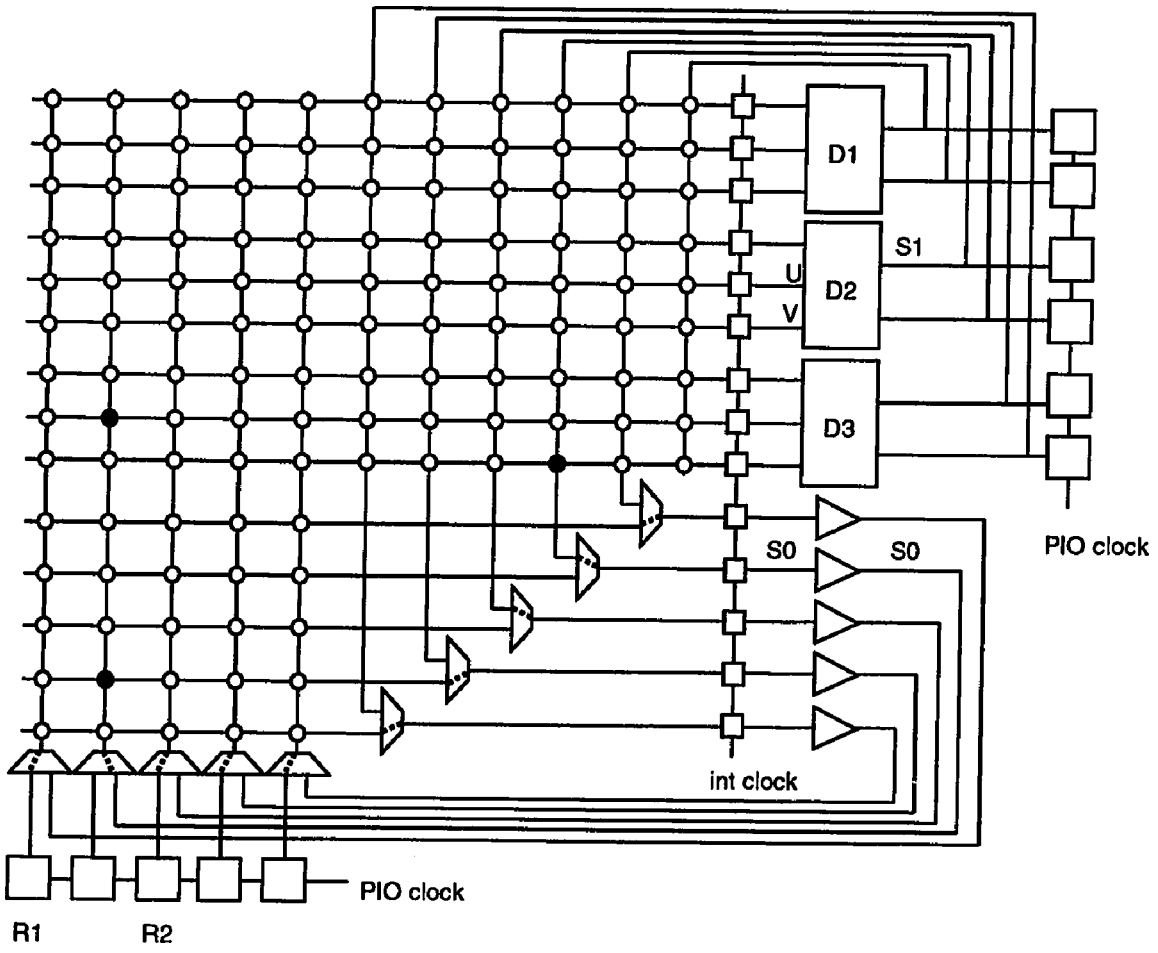


Figure 9.16: Sequential Example. DIA: Level 2 → Level 3; DLMs: Level 2.

Chapter 10

Variations and Modifications

In Chapter 8 we presented the *Dharma* architecture. The detailed block diagram (Figure 8.6) is just one possible way of realizing the dynamically reconfigurable folded pipeline of Figure 8.5. In fact, Figure 8.5 can be thought of as a generic architecture. Several variants to the specific architecture of Figure 8.6 are presented in this chapter. Variations and modifications of the DLM are grouped together in one section; similarly variations and modifications of the DIA are grouped together in another section. The DLM variations and the DIA variations are independent, unless stated otherwise. Therefore, each of the DLM variations can be used with each of the DIA variations; further, some variations within a group can be used along with the other variations in the same group. Hence, a large class of dynamically reconfigurable architectures is possible (see shaded region of Figure 1.3). The last section in this chapter describes how certain circuits with number of topological levels greater than the maximum number of levels allowed on a single *Dharma* chip, can be implemented on one chip, by means of modifications to the generic architecture.

10.1 Dynamic logic module

In Figure 8.6, we have shown the DLM to be a K-input, 1-output lookup table. *Dharma* is not necessarily limited to this kind of logic module. In general, any logic structure, which can be repetitively changed to implement different functions, can replace the K-input, 1-output LUT. In this section, we present complex LUT DLMs, multiplexer based DLMs, PLA based DLMs and heterogeneous DLMs.

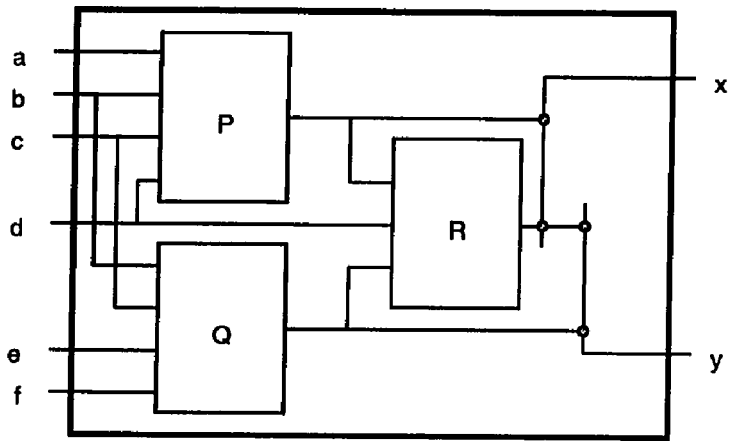


Figure 10.1: An example of a complex LUT DLM

10.1.1 Complex LUT DLMs

The K-input, 1-output LUT is one specific, simple manifestation of a broader class of reconfigurable logic modules based on the lookup table. A K-input m-output ($m > 1$) LUT and interconnected LUTs are other such logic modules in this class; and can replace the K-input 1-output LUT in the *Dharma* architecture.

As an example, Figure 10.1 shows one possible complex DLM with 6 inputs (a, b, c, d, e and f) and 2 outputs (x and y). P and Q are 4-input 1-output LUTs and R is a 3-input, 1-output LUT. P and Q share two inputs (b and c) and P and R share one input (d). Any two of P, Q and R can be selected as the DLM outputs by means of a mini-switching network on the output lines. The black dots represent crosspoints, which are similar to the crosspoints in a crossbar. Each output line has to be connected to at least *and* at most one of the LUT outputs. Hence exactly one switch on each output line is going to be on, and the other is going to be off. This implies that only one bit is required per output line, when the bit is set to '1', it turns on one of the crosspoints, and when it is set to '0', it turns on the other. P and Q require 16 bits each, R requires 8 bits and the crosspoints require 2 bits – giving a total of 40 bits per level of logic.

Several other LUT interconnections are possible, Figure 10.1 is meant to only give a feel for what is possible. Xilinx CLBs [Xilinx 89] and the ORCA PLC [ORCA 93] are

good examples of complex DLMs [1].

10.1.2 Multiplexer based DLMs

The DLM is not restricted to use only LUTs. Like the Actel [Actel 91] logic module, the DLM can be constructed from multiplexers. Unlike the Actel logic modules, the programmable elements will be read/write memory bits in *Dharma*, and not anti-fuses.

10.1.3 PLA based DLMs

A two level AND-OR logic plane can be used for the DLM. That is, a mini-PLA or mini-PAL or any such other programmable AND-OR block can function as the reconfigurable logic module. Figure 10.2 shows a symbolic diagram of a 5-input, 2-output DLM with programmable AND plane and programmable OR plane. There are 5 product terms, and each output OR gate has these 5 terms as inputs. The inputs enter the AND plane in both true and complement form. The connections from the input lines to the AND gates, and the connections from the AND gate to the OR gate are programmable, and are controlled by means of stored configuration bits. An input can be present in either true or complement form (and not both) in any product term, or it need not be present at all. This implies that each intersection point requires a bit (i.e., we cannot use the bit saving scheme that was used in the mini-crossbar of Figure 10.1). For the example shown in Figure 10.2, the AND plane requires 50 bits, and the OR plane requires 10 bits, giving a total of 60 bits per level of logic.

Several other programmable AND-OR types of circuits (eg., fixed OR plane, invertible outputs, etc.) are possible, and Figure 10.2 serves only as an illustration of what we mean by a PLA based DLM.

10.1.4 Heterogeneous DLM array

The above discussion assumes that all DLMs in the logic module array (see Figure 8.5) are identical. This need not be the case. The array can be composed of different types of DLMs, with a couple of each type. The logic module array in Figure 10.3 has 5 DLMs, 2 of which are 3-input AND gates, 2 are 5-input 1-output LUTs and one is an

[1] Only the combinational part of the CLB or PLC can be used as a DLM. The latches in the CLB and PLC do not serve any purpose in the DLM.

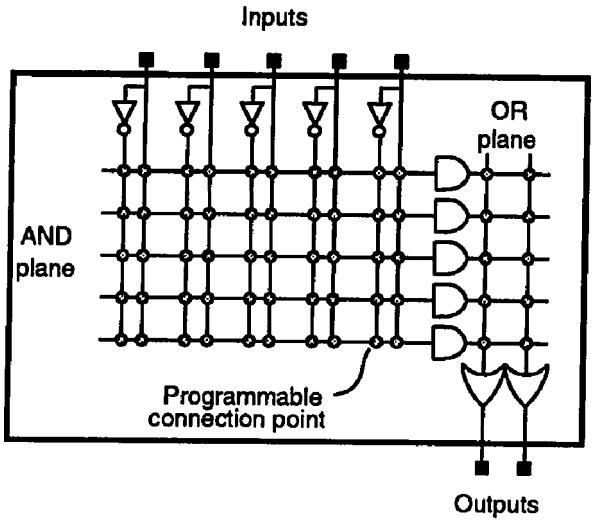


Figure 10.2: An example of a PLA based DLM

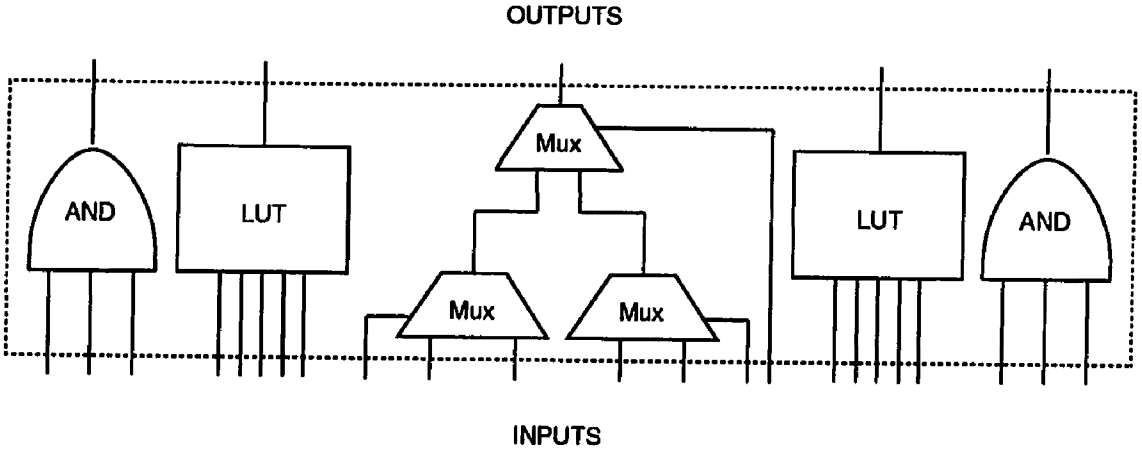


Figure 10.3: A heterogeneous DLM array

interconnection of 2-input multiplexers. The 3-input AND gate is not reconfigurable, and may seem out of place, but usually many combinational functions are simple ANDs or ORs, and by having these fixed functions in the DLM array, it is possible that there will be a saving in silicon area.

10.1.5 Technology dependent modifications

Modifications are also possible in how the actual implementation of the DLM is made. eg., a K-input, 1-output lookup table can be implemented using SRAM, DRAM, etc., technology. The dynamic nature of the DLMs is especially attractive to DRAM technology, since the refresh rate of the memory cells can be synchronized with the level addressing. Real-time reconfiguration does not imply that the memory has to have a 'write' capability. Read-only memories can also be used to store the configuration bits, the appropriate bits being chosen at each level by proper addressing using the level bits.

10.2 Dynamic interconnect array

The dynamic interconnect array has a large number of variations possible, both in terms of the overall architecture, and the actual implementation thereof. We have discussed below many of such variations.

10.2.1 Shift register at crosspoint

Each crosspoint of the crossbar has L bits of memory, a bit being used per level. These bits could be arranged in the manner of a shift register so that the change of level would just mean shifting one bit. Figure 10.4 shows a portion of the L-crossbar which uses this scheme. The connection at each crosspoint is achieved by means of an NMOS pass transistor, and each shift register has 4 bits (i.e., $L = 4$). Lines a, b and c enter the crossbar from the PI muxes (see Figure 8.6), and the output lines x and y go to the latches at the DLM inputs. Connection between the horizontal and vertical lines is established whenever a '1' is present at the gate of the pass transistor.

The shift registers are clocked by the internal clock, and at the clock-edge all bits shift by one to the right. The rightmost bit loops back to the first bit. Figure 10.4(a) shows the bits at level d, and Figure 10.4(b) shows the bits shifted right by one, for the next

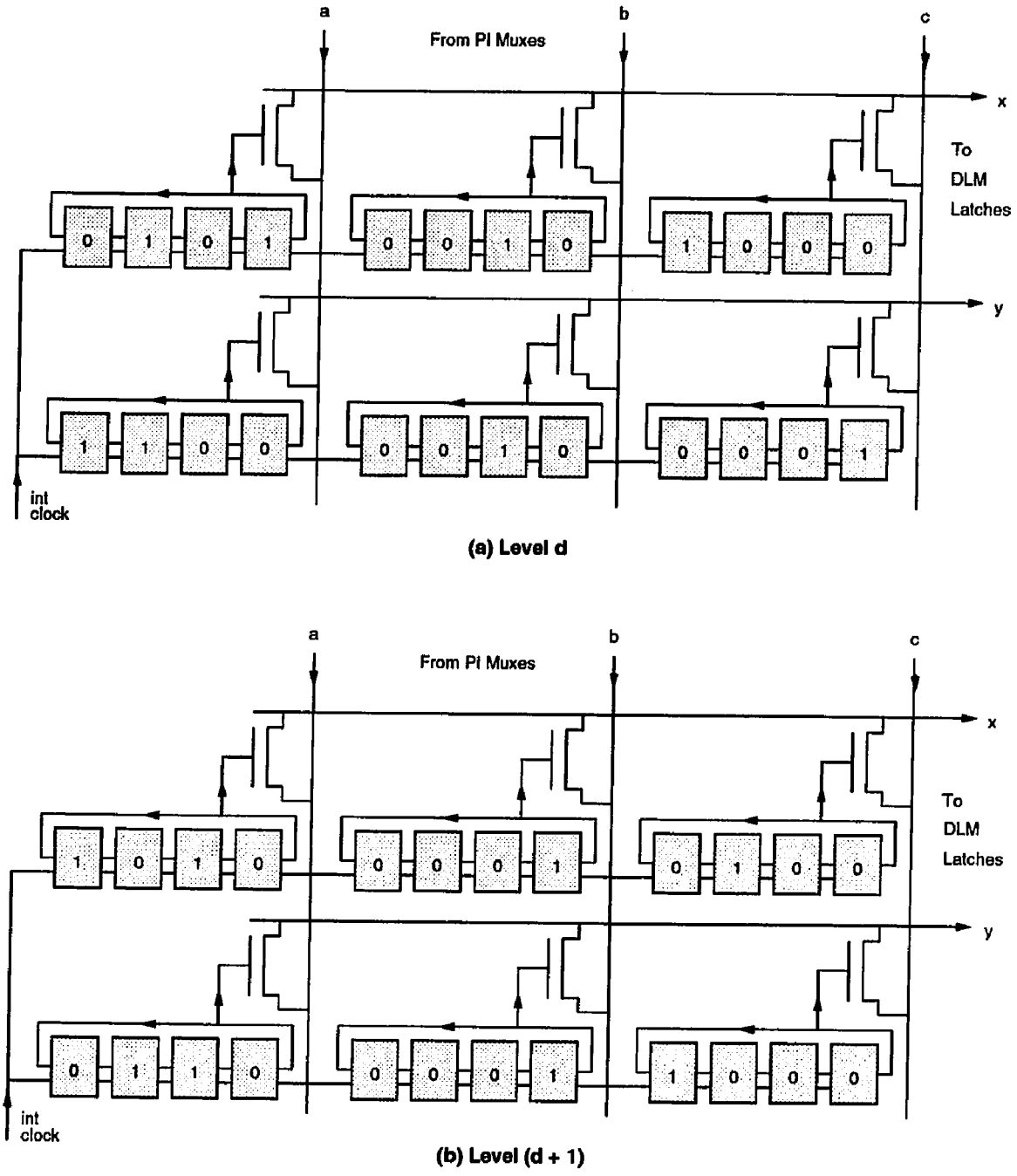


Figure 10.4: An example of L-crossbar which uses shift registers

level $(d+1)$. In (a), vertical line a is connected to horizontal line x, and vertical line c is connected to horizontal line y. Similarly, in (b), b is connected to x and y [2].

The disadvantage in such a shift register based DIA would be that the entire L bits will have to shift through before returning to the first level, and hence only circuits with l levels, where l is a perfect divisor of L can be implemented. However, this may not pose any real problem. The chip manufacturer can provide a series of chips with different L. The user then has to pick the chip with the appropriate L (i.e., L such that it is equal to the circuit's levels l or is an exact multiple of l). For example, a chip with $L = 6$ can implement a circuit having $l = 2$, 3, or 6. The $l = 2$ and $l = 3$ circuits, however, will be wasting a third, and a half of the resources, respectively.

10.2.2 Multiplexer based crossbar

Assume we are using a K input, 1-output LUT as a DLM. Let there be C such DLMs and B pass-buffers. Since there are $(B+C)KC$ crosspoints in the L-crossbar, we will require $(B+C)KCL$ memory bits. However, a signal fans out to at most one pin of the DLM. In other words, the $(B+C)$ signals need to connect to only one of the K inputs per DLM. This means that the K bits can be reduced to $\log_2 K$ bits per signal, reducing the total bits to $(B+C)(\log_2 K)CL$. However, this would warrant a decoder at every DLM, for each of the $(B+C)$ lines. A more intelligent way of designing the crossbar is to recognize that at each LUT input we need, not a $(B+C)$-to-K crossbar, but a $(B+C)$-to-K multiplexer. A $(B+C)$-to-1 multiplexer requires $\log_2(B+C)$ select lines, and we require K such multiplexers. Each select line needs a bit per level, therefore we need $KCL\log_2(B+C)$ bits and KC $(B+C)$-to-1 decoders. This would be more area efficient if

$KCL\log_2(B+C) + KC(B+C)\delta < KCL(B+C)$, where δ is the bit-equivalent of a decoder and gate.

i.e., $\delta < L(1 - \frac{\log_2(C+B)}{(C+B)})$.

Since the DLMs and interconnections both require decoders, there could perhaps be a sharing of decoders possible. This could require extra latches at the DLM outputs, and could also increase the routing delay.

[2] It is an error to have more than one vertical line connected to a horizontal line in the same level.

10.2.3 Partially hard-wired DIA

A further decrease in the number of crosspoints can be achieved if we allow certain hard-wired connections. Each DLM can have a hard-wired connection to K DLMs, and each DLM will have hard-wired inputs from K DLMs, preferably those that are adjacent to it. This would imply that the crossbar (or multiplexer, as the case may be) would then be reduced to a $(B + C - K) \times KC$ crossbar, and there would be KC additional 2-input multiplexers (one at each DLM input), with associated KCL select bits. This interconnection scheme would decrease the number of fanout points from the DLM output to $(C - K)K + K$, instead of the earlier KC. However, the delay in the interconnect could increase, since the signal has to pass through an additional multiplexer (2-to-1 multiplexer).

Figure 10.5 shows the hard-wire connections from DLM *D3* to its adjacent DLMs. $K = 5$ in this figure. The output of *D3* is *hard-wired* (through the 2-to-1 multiplexer) to input 5 of *D1*, input 4 of *D2*, input 3 of *D3*, input 2 of *D4* and input 1 of *D5*. The output of *D3* is connected to other DLMs (other than *D1* ... *D5*), through the L-crossbar. Figure 10.6 shows a portion of the L-crossbar corresponding to the example of Figure 10.5. As a result of the hard-wire connections, several crosspoints are now unnecessary. For example, the output of DLM *D3* has no crosspoints at the input lines corresponding to DLMs *D1* ... *D5*. If *D3* needs to connect to *D1*, for example, then the hard-wire connection must be used.

10.2.4 Binomial concentrator based DIA

A different type of interconnection strategy can be adopted for LUT DLMs based on the following observation: the DLM inputs are all equivalent. Hence, the interconnection switch at the inputs is, by circuit switching terminology, a $(B + C)$-to-K concentrator. In other words, we need to select any K of the $(B+C)$ signals. This can be achieved by using a two-level sparsely connected crossbar, called a binomial concentrator [Masson 77]. The chief advantage of using such an interconnection strategy is that the number of crosspoints are reduced and as a result the number of bits used for storage are reduced. The disadvantage is that the signals now have to pass through two crosspoints in series. This could increase the interconnect delay (this is not clear at this point, simulation studies have to be performed to arrive at a definite answer – the reduced crosspoints implies that the signals have to be connected to lesser switch inputs, and this will reduce the parasitic capacitance, and could improve speed).

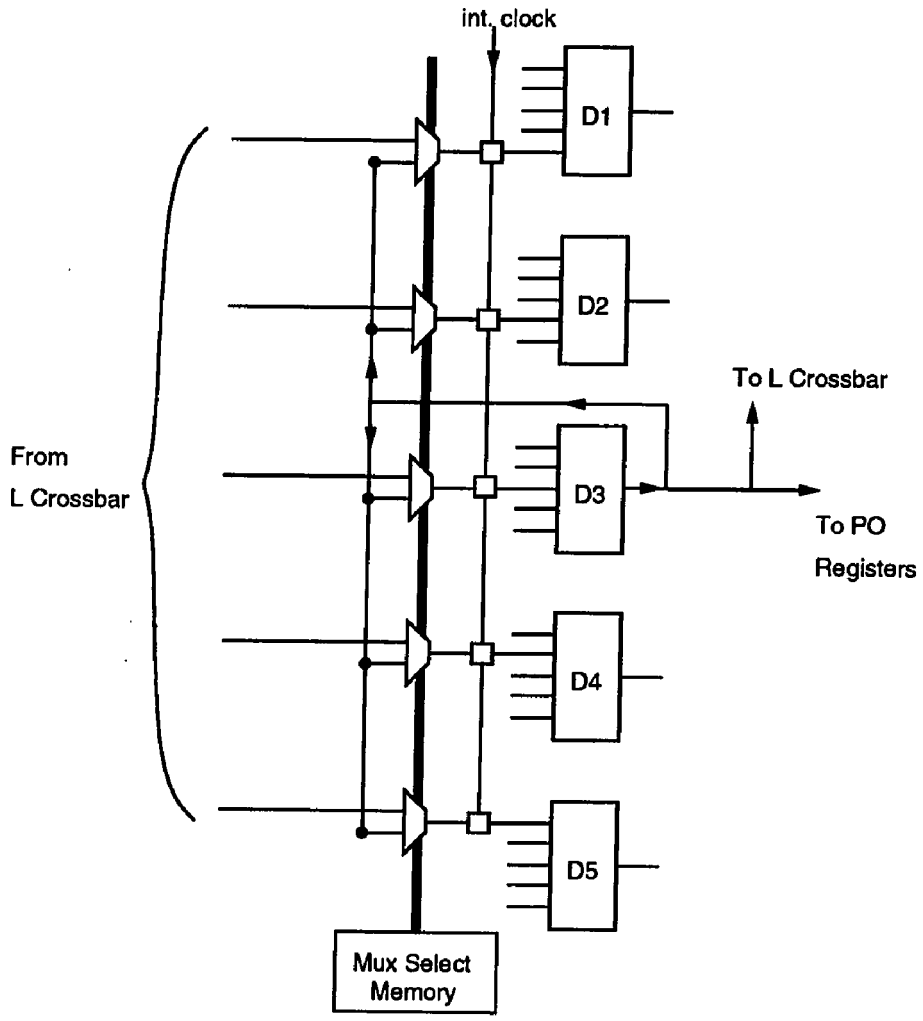


Figure 10.5: Hard-wire connections from a DLM to adjacent DLMs.

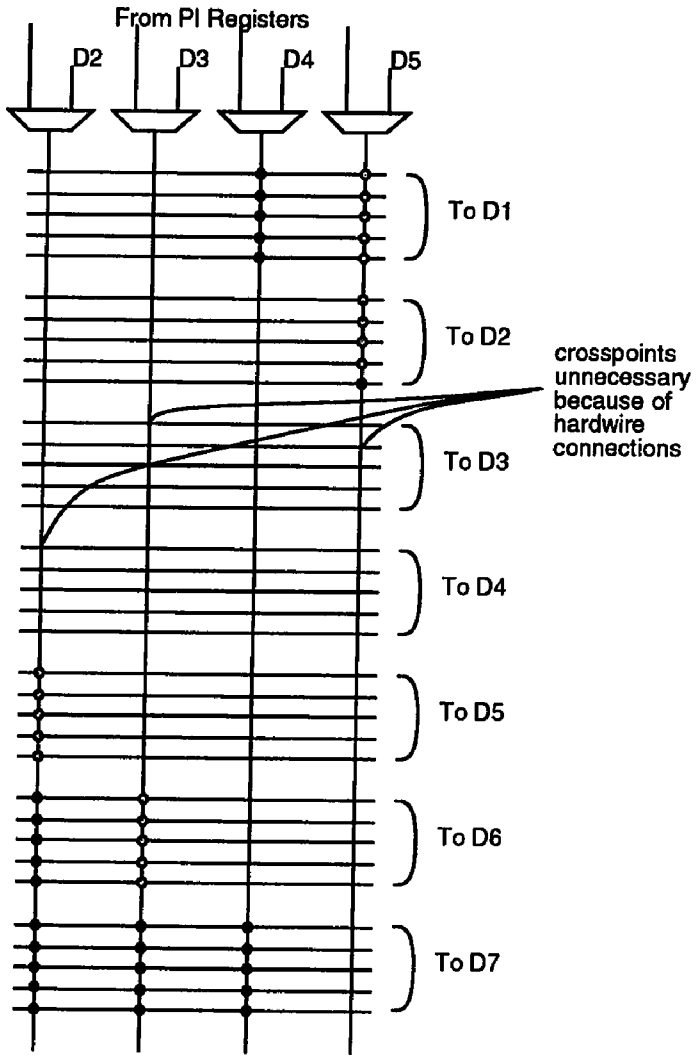


Figure 10.6: Hard-wire connections reduce L-crossbar crosspoints.

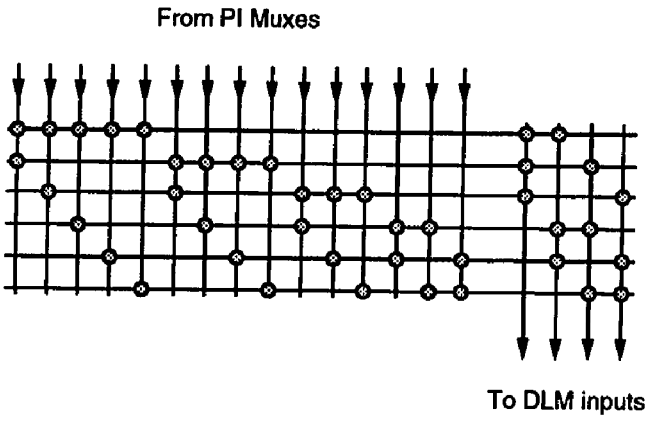


Figure 10.7: A 15-to-4 binomial concentrator

Figure 10.7 is a 15-to-4 binomial concentrator. This could be used as a part of the L-crossbar when $K = 4$. The crosspoints are distributed in such a manner that any four of the DLM and pass-buffer outputs can be connected to the DLM input lines.

10.2.5 DIA as a *Copier*

The above discussion has centered on using the crossbar to implement a real connection, i.e., the interconnect switch connects its inputs to its outputs (say through transistors). This type of connection is *required* in the case of analog signals, where the input signals can have different values. However, in our case we are dealing with digital signals, which take on only two values, either 0 or 1. So, the switch will still be functioning properly if its outputs are set to the value of the inputs they are supposed to be connected to. In essence, the switch is required to perform a one time copy of a selection of the inputs (those that are to be connected to the outputs) to the outputs. Can we take advantage of this fact to speed up the connection and reduce interconnect area? One answer to this question is delineated in the next paragraph.

Consider the case of just one switch output (DLM input). It can be connected to any one of the $(B + C)$ signals (switch inputs) coming from the previous level. That is, its value should be set to the value of one of the $(B + C)$ signals. This is equivalent to a memory read operation, where the memory consists of $(B + C)$ bits, and one of them has

to be read onto the switch output. So, the selection can be performed by using precharge circuitry and sense amplifiers. The $(B + C)$ signals correspond to one column of memory, and one of the cell select transistors is chosen by means of a decoder or local memory bit. There are both area and speed advantages in using this style of implementation. The cell select transistors are of minimum size as opposed to the large pass transistors that would have been required if a direct connection between switch input and output were required. And using the sense amp, the read operation can be performed fast. Further, the precharge time of the memory can be interleaved with the DLM propagation time, thus shortening the interconnect delay even further. This would mean that the interconnect delay would be of the same magnitude as an LUT based DLM (since the logic module delay is also a memory read operation). It could be even less, since the logic module is a $2^K L$ bit memory (in the case of LUT based DLM) whereas the interconnect is a $(B + C)$ bit memory[3].

Figure 10.8 shows a section of the L-crossbar which uses this idea. The transistors shown in this figure can be much smaller (and are usually made of minimum dimensions) than those in Figure 10.4. The small squares sitting on the gate of the transistors represents the configuration bit at a given time instant (this can be presented at the gate using the shift register scheme, or the regular scheme of Figure 8.6). These are gated by the internal clock, so that they will be connected to the gate only after the vertical lines have a valid signal on them. All horizontal lines are precharged simultaneously. At the clock, the transistors with a '1' on their gate will try to equalize their respective horizontal line's voltage to that of their vertical line, and this causes a swing in the voltage on the horizontal line. The sense amplifiers detect the direction of the swing and set their output (going to the DLM inputs in Figure 10.8) to a '0' or '1' accordingly. This precharge/sense scheme requires that the voltage on the (highly capacitive) horizontal lines swing by only a tiny amount, which causes the propagation delay through the crossbar to be small. For the values shown in the figure, c gets connected to x and z, and b gets connected to y.

10.2.6 Sparsely connected K-class DIA

So far, we only discussed fully connected switches, where it is always possible to connect any switch input to any switch output. These switches tend to be expensive in terms of area. It could be possible to decrease the area requirement of the DIA, by giving

[3]There are KC such $(B + C)$ bit memories.

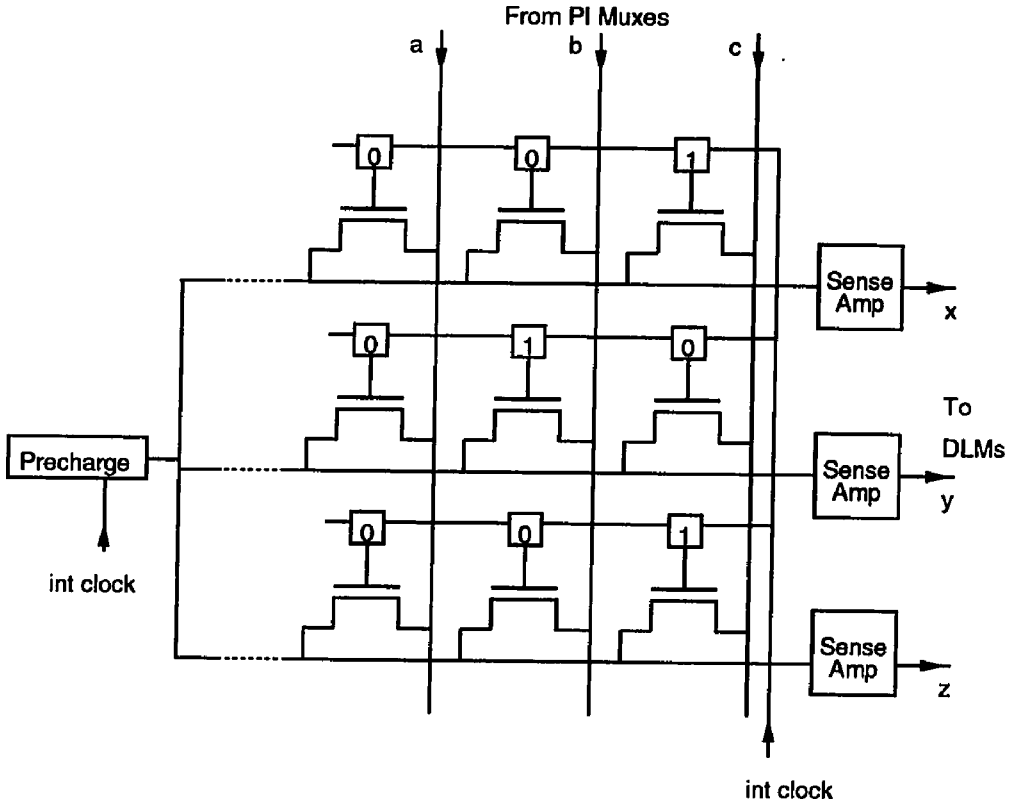


Figure 10.8: DIA as a *Copier*

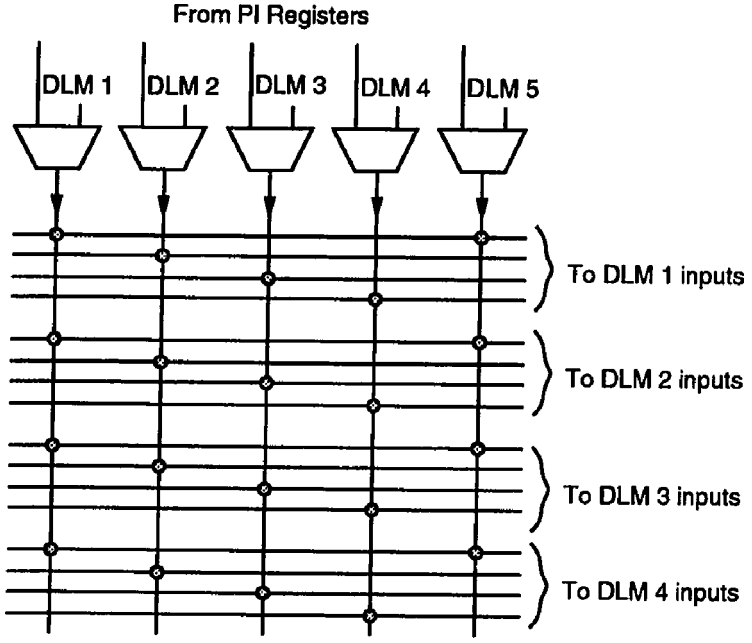


Figure 10.9: K-class DLMs reduce crosspoints

up the full connection flexibility. We discuss below a smart way of reducing the number of crosspoints, while maintaining a high probability of routing completion.

This scheme requires that the DLM have all inputs equivalent, as in the case of the LUT. Assume there are C DLMs, each with K inputs and 1-output. We divide the DLMs into K classes. Each class will have at most $\lceil C/k \rceil$ and at least $\lfloor C/K \rfloor$ modules. A module in class j, $j = 1 \ldots K$, is connected to the jth input of each DLM (instead of to every input). This will decrease the number of crosspoints and memory bits by a factor of K. It has been found that for all the benchmark circuits tested, it is always possible to achieve complete connectivity using this connection strategy. Note that, we could still have some DLMs connected to every input, just in case it is not possible to complete the connections.

Figure 10.9 shows a portion of the L-crossbar which uses the sparsely connected K-class method. Here $K = 4$. The outputs of DLM 1 and 5 connect only to the first input lines of all DLMs; similarly output of DLM 2 connects only to the second input lines; and

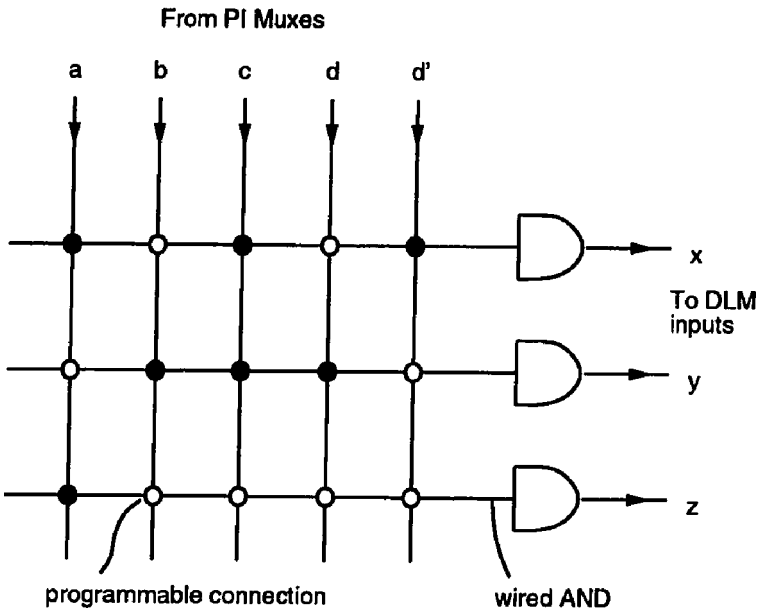


Figure 10.10: A DIA with wire-AND capability

so on.

10.2.7 Functional DIA [4]

In addition to performing a connection, it is possible to design the DIA such that a logic operation, such as AND or OR, can be performed together with the connection.

Figure 10.10 explains this concept. A portion of the L-crossbar is shown. The crosspoints are designed such that when more than one vertical line is connected to the horizontal line, an ANDing operation takes place. The connections (from vertical line to horizontal line) are programmable, as before. In the figure, a dark circle at the crosspoint represents a connection. Therefore,

$$x = acd'$$
$$y = bcd$$
$$z = a$$

[4] Dr. Kamal Chaudhary originated this idea.

Each wired AND gate can have $\leq (B + C)$ inputs, thereby making this scheme very powerful. Since every logic function can be expressed in the sum-of-products form, the products (i.e, ANDing) can be formed in the DIA, and the sum (i.e., ORing) can be performed in the DLM. In other words, the DLMs can just be OR gates. As a result of the wired ANDing, it has been possible to reduce the number of levels in the circuit by 15-20% [Chaudhary 93].

To realize the full potential of this scheme, the signals may have to enter the crossbar in both true and complement form (eg., d and d'). This increases the size of the crossbar. Also, to achieve the wired AND, the number of transistors at each crosspoint will be doubled, compared to the normal case.

10.2.8 Technology dependent modifications

Like the DLM, the DIA is also easily suited to many different technologies. For example, if one were to use a DRAM technology, there would be considerable savings in area, without any complexity increase, since the DRAM refreshing could be synchronized with the level addressing. Like the DLM, a non-volatile memory technology (ROM) can also be used for the DIA. The shift register scheme can benefit in area if a technology such as a charge-coupled device (CCD) is used.

10.3 Level expander

Consider a feasible network $\mathcal{N}$ (i.e., each node has $\leq K$ inputs) with S nodes, and l levels. We wish to implement this circuit on a *Dharma* chip with C DLMs, each with K inputs, and L levels. If $CL \leq S$ but $l > L$, then the chip has the resources (in terms of total logic modules, and interconnection configuration bits) to implement $\mathcal{N}$, but the architecture of Figure 8.6 needs to be modified to allow this. We call this modification the *Level Expander* circuitry. By using this modification, the number of levels that can be implemented in an L level chip can be raised to Γ, $\Gamma > L$, without increasing the crossbar and DLM configuration memories (the PI Mux and Buffer Mux select memories will need to be enlarged).

10.3.1 Example

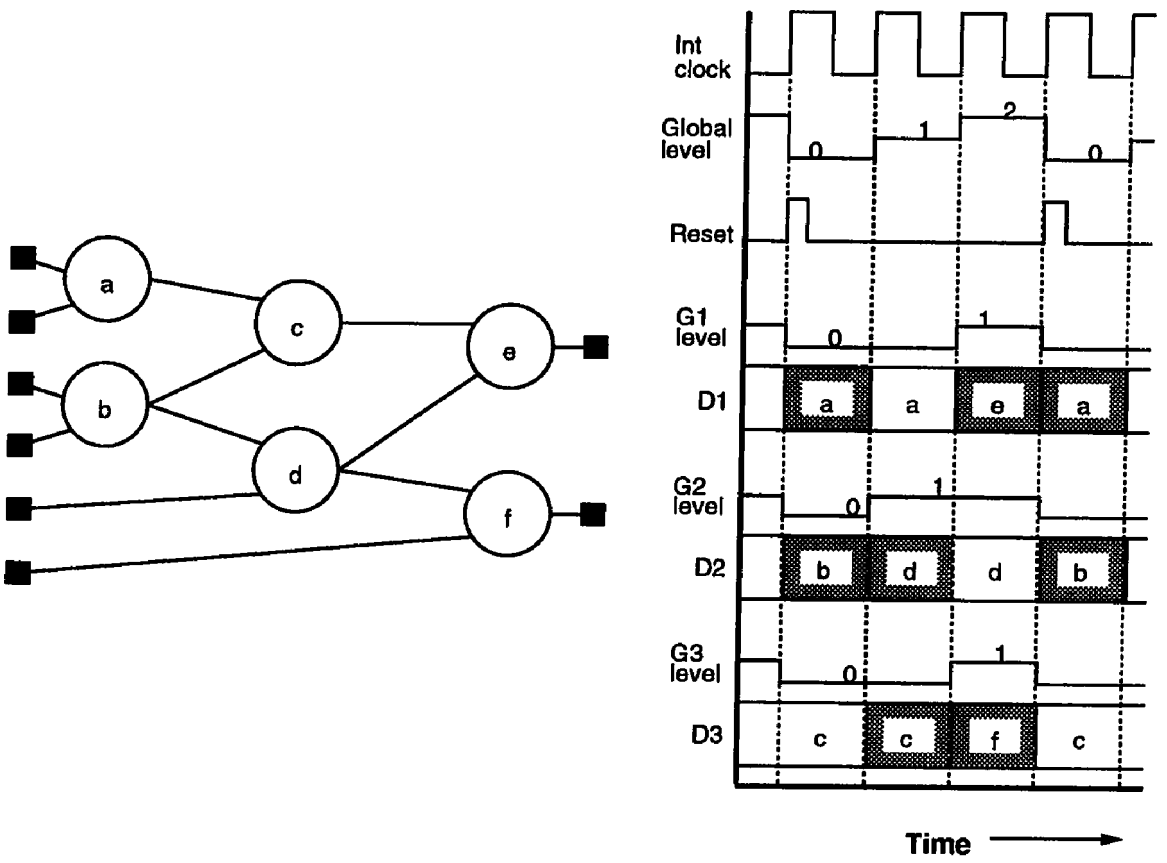


Figure 10.11: An example to illustrate level expansion

The principle of level expansion is best illustrated by means of a simple example. Let us consider a tiny *Dharma* which has 3 DLMs, and has $L = 2$. We wish to implement the DAG of Figure 10.11 on this tiny chip. The DAG has $S = 6$ nodes, but $l = 3$. Therefore, $CL = S = 6$, and $l > L$. Without level expansion, it would not be possible to implement this DAG on the tiny *Dharma*. The circuit has only 2 nodes per level, although the chip has 3. This implies that 1 DLM would go wasted per level, leaving us with 2 wasted DLMs, and 2 unimplemented nodes. The level expansion idea is to assign the unimplementable nodes to the wasted DLMs. To see how this is done, consider the rows *D1*, *D2* and *D3* of the waveform diagram in Figure 10.11. Ignore the rest of the waveforms for the present. In the first time step, DLMs *D1* and *D2* perform functions a and b, respectively. In the second time step, DLMs *D2* and *D3* perform functions d and c, respectively, and in the third time step, DLMs *D1* and *D3* perform functions e and f, respectively. This assignment ensures that each DLM performs only 2 functions (since each DLM has resources for only 2 levels), and hence the circuit of Figure 10.11 can be realized on the tiny *Dharma*. To allow this to happen, however, DLMs *D1* and *D3* must be prohibited from implementing their second function (i.e, e and f, respectively) in the second step. Similarly, DLM *D2* must be prohibited from changing back to its first function in the third step. This will not be possible if all the DLMs are addressed by the same level bits. Therefore, each DLM needs its own *local* level counter. The level bits of the local level counter address the DLM configuration memory, and the crossbar memory at its inputs.

10.3.2 Expander circuitry

Figure 10.12 shows the related circuitry to make this possible. Compare this figure with Figure 8.6. The L-crossbar dynamic configuration memory (shortened to *DCM* in Figure 10.12) has been partitioned into n parts, where n is the number of DLMs. Each partitioned memory (*L1*, *L2*, ...) addresses the crosspoints at the inputs of a single DLM (*D1*, *D2*, ..., respectively). Similarly, the B-crossbar DCM is also partitioned into m parts (*B1*, *B2*, ...), where m is the number of pass-buffers. This partitioning scheme does not alter the number of memory bits, or routing resources. Each DLM and associated DCM are addressed by local level bits, generated by a *Local Level Generator*. Each buffer DCM is also similarly addressed. Hence there are a total of $(n + m)$ local level generators. The *Level generation circuitry* block of Figure 8.6 now functions as a *global* level generator. The level

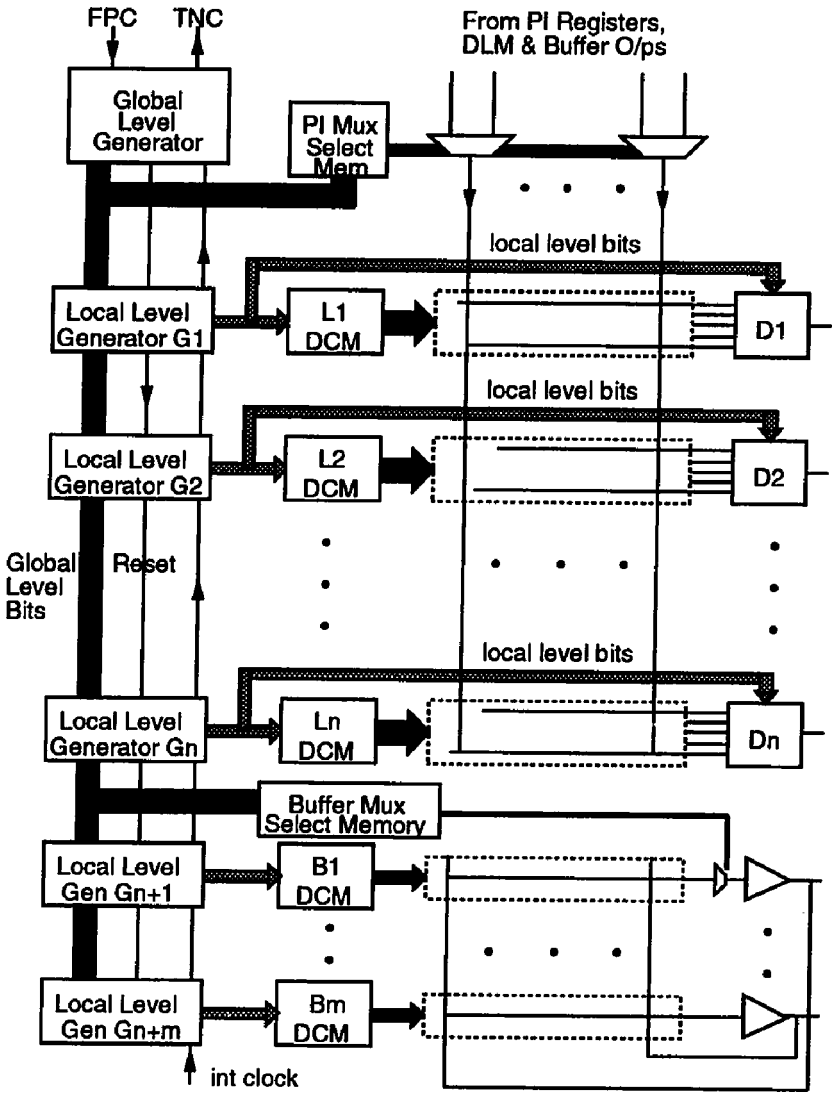


Figure 10.12: Modified *Dharma* with level expansion capability

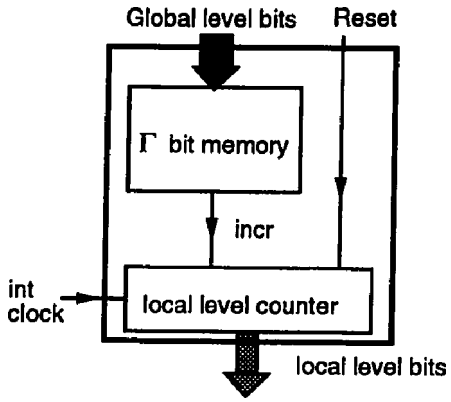


Figure 10.13: The *Local Level Generation* circuitry

bits generated by it are termed global level bits. Let Γ be the maximum number of circuit levels that can be implemented on the chip. Then, the global level counter (i.e., the level counter of Figure 8.11) is a divide-by-Γ counter. The local level generators are addressed by the global level bits. The *Reset* signal of Figure 8.11 is now made globally available to all the local level generation blocks.

The multiplexer select memories (PI Mux select and Buffer Mux select) are addressed directly by the global level bits, and these have to be enlarged to Γ bits each. Since each mux is individually selected, the local level generation scheme will increase the number of bits used, rather than provide a saving, and hence the original addressing scheme is better in this case.

Not shown in Figure 10.12 are the input latches. These have to be clocked only when the DLM function is to change in the next clock, and hence their clock has to be gated by the *incr* bit of their respective local level generator.

10.3.3 Local level generator

All the local level generators are identical. Figure 10.13 shows a detailed diagram of the circuitry. Each local level counter is a divide-by-L counter. Each local level generator has Γ bits of control memory that is addressed by the global level bits. One bit of memory is output every clock cycle (labeled *incr* in the figure). The local level counter is incremented

at the clock, when *incr* is asserted. Going back to Figure 10.11, the global level generation circuitry broadcasts a reset signal every 3 cycles, since the circuit being implemented has 3 levels. The local level counters in *G1* and *G3* hold their count at 0 in the second step, and the local level counter in *G2* holds its count at 1 in the third step. The *incr* bit is de-asserted in their respective control memories to make this pause in the count possible. Note that the function c computed by DLM *D3* in step 1 (which gives an erroneous value for c, since a and b are not yet ready) is ignored, since c is computed again in step 2. Although Figure 10.12 and Figure 10.13 may imply that the Γ bit control memories are distributed, in actual implementation, it can be single block of memory with $(m+n)$ bit outputs going to the various local level generators.

Chapter 11

Synthesis for Dharma

This chapter deals with mapping designs onto *Dharma*. We consider only the basic architecture of Chapter 8, and assume that the DLM is a K-input, 1-output LUT. The first section gives an introduction to what is required in a synthesis tool for *Dharma*. We look at one aspect of the synthesis, namely, the *temporal partitioning*[1] problem, where we assign nodes in the input network to levels (i.e., we assign a time slot to each). This problem can be formulated as an integer programming problem; the second section shows how that is so. However, we do not use the integer programming problem to do the level allocation - instead we solve it by using a heuristic algorithm; that is the subject of the third section. The last section presents results for the heuristic approach.

11.1 Introduction

Let us consider a *Dharma* chip with L levels and C DLMs. There are two aspects to the synthesis for *Dharma*. On one hand we need to generate a circuit such that the number of levels in the circuit l is minimized, since the worst case delay is proportional to l (see Chapter 12). On the other hand, the maximum number of nodes in any level, G, should be such that $G \leq C$, otherwise the circuit cannot be implemented on a single chip.

We tackle this problem in a two-step manner.

1. Given an input circuit specification, transform it into a DAG such that the number of inputs at every node $\leq K$, and the maximum number of topological levels l is minimized.

[1] Credit goes to Dr. Rajiv Shah for coining this term.

2. Allocate levels to the nodes obtained in Step 1, in a manner such that $G \leq C$ (*temporal partitioning*).

Techniques for doing Step 1 are already known, for the case when the DLM is a LUT (see [Cong 92, Murgai 91a, Francis 91a] and Chapter 5). Hence we concentrate only on step 2. Note that it may not always be possible to satisfy the condition that $G < C$ in Step 2. If so, we allow l to be increased. In Chapter 8, it was suggested that the input DAG be topologically levelized to allocate nodes to DLMs. Topological levelization is one method of doing the level allocation, but there are other methods too. From the requirements of step 2, topological levelization may not give a satisfactory result, since the levelization process does not consider the $G < C$ constraint. Hence, the need for a different levelization procedure.

Before discussing the details of the new levelizing scheme, we first define what exactly we mean by topological levelization.

11.1.1 Topological levelization

There are two ways of doing topological levelization.

Starting from the primary inputs, the levelization proceeds in the following manner.

Primary inputs are assigned level 0.
Level of a node = MAX(level of its fanins) + 1

Figure 11.1(a) shows level allocation using this algorithm for a very simple DAG example (although not shown in the figure, the edges are directed, and are from left to right).

Levelization from the primary outputs is similar, and is described below (for a DAG with M levels). Starting from the primary outputs, the levelization goes as follows.

Primary outputs are assigned level M .
Level of a node = MIN(level of its fanouts) - 1

Figure 11.1(b) shows the level allocation using this algorithm for the same DAG of Figure 11.1(a). Note that it is possible, and is usually the case, that the two level assignments will be different. Only nodes on the maximum path (i.e., a path from input to output with maximum length, length measured in terms of number of nodes) will have the same level assigned by both schemes. In the figure, nodes c and d are assigned to different levels by the two schemes. We call these nodes *slack nodes*, since there is some slack allowed in their level assignment. Slack nodes cannot lie on maximum paths.

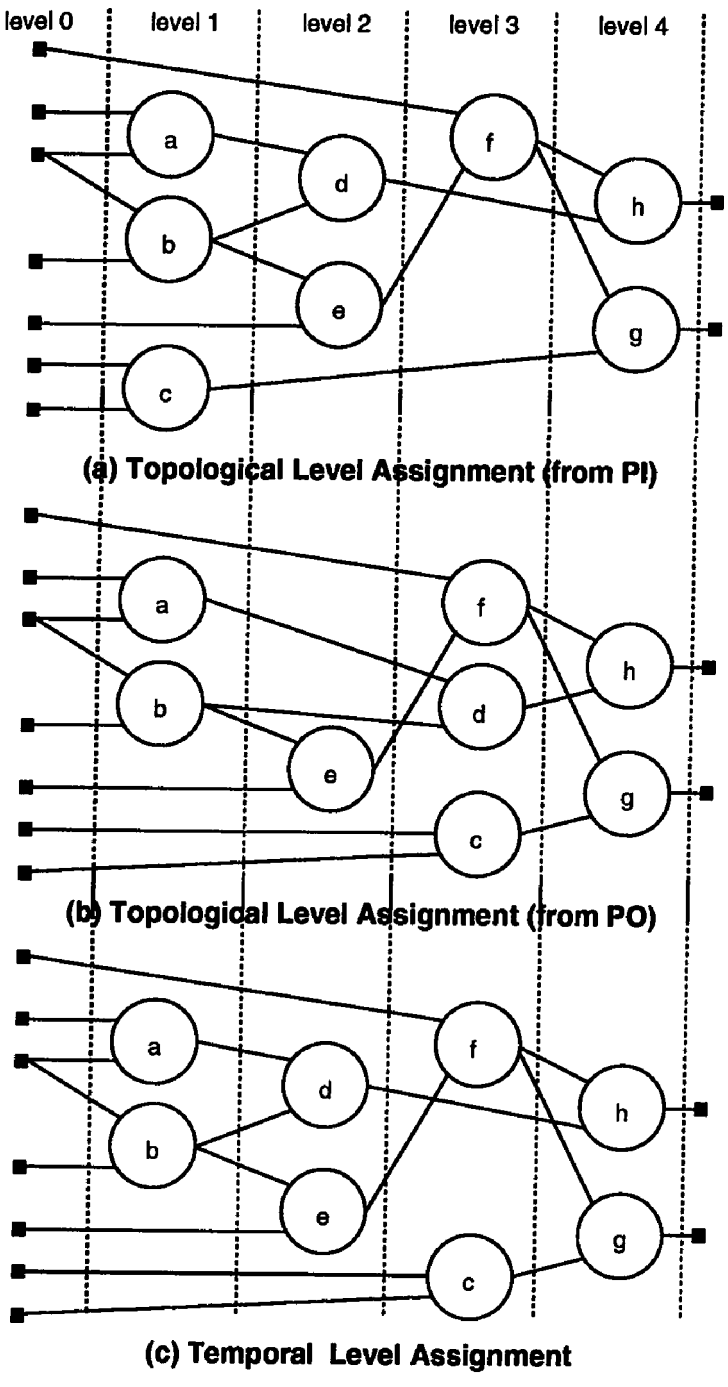


Figure 11.1: Different levelization techniques

11.1.2 Temporal partitioning

We need to determine which node functions can be evaluated in parallel, and at what time instant. That is, the nodes have to be partitioned in *time*. Hence the name temporal partitioning. More simply, each node has to be allocated a level, which could be different from its topological level. This cannot be solved by simple topological levelization always, as will be apparent from the discussion below.

Assume that the logic network is a feasible network (i.e., each node in the network has $\leq K$ inputs). Let the combinational part of the feasible network be topologically levelized. Let l be the number of levels, S the total number of modules and G the maximum number of modules in any level. Note that G is dependent on the levelization scheme used. Let $t_l(n)$ be the topologically assigned level of node n, and let $a_l(n)$ be its allocated level, which will be assigned by the temporal partitioner.

If $l \leq L$ and if the topological levelization has $G \leq C$, then there is no need for temporal partitioning, since each node can be assigned a level that is the same as its topologically assigned level, i.e., $a_l(n) = t_l(n), \forall n$.

If, however $l \leq L$, and topological levelization has $G > C$, with $S \leq CL$, it may still be possible to implement this circuit on *Dharma* by a judicious choice of levels for the *slack nodes*. These nodes can be assigned a new level (different from their topological level); i.e., they are *moved* in *time*.

Consider the DAG of Figure 11.1. Assume it is to be implemented on a tiny *Dharma* chip which has $C = 2$ and $L = 3$. The DAG has $S = 6$ and $l = 3$. The topological level assignment from primary inputs (Figure 11.1(a)) has $G = 3$ (level 1 has three nodes a, b and c), and the topological level assignment from primary outputs also has $G = 3$ (level 3 has three nodes c, d and f). This may suggest that it is not possible to implement this DAG on the tiny *Dharma*. However, the level assignment in Figure 11.1(c) has $G = 2$, and this implies that it is indeed possible to implement the DAG on our tiny *Dharma*. Note that the levelization of Figure 11.1(c) can be derived from Figure 11.1(a), by moving c alone to level 3, and leaving the rest of the nodes unaltered. Similarly, we can derive Figure 11.1(c) from Figure 11.1(b), by moving node d to level 2, and leaving the other nodes unchanged.

The process of identifying nodes to be moved, and moving them appropriately in an effort to keep G within C, is what we call temporal partitioning.

11.1.3 Handling other designs

If $l > L$, with $S \leq CL$, then it may still be possible to implement the circuit on *Dharma* by grouping together modules across several levels. Each such group can then be realized in more than one internal cycle, before switching over to the next group. This has been discussed at length in Section 10.3. Techniques to choose which nodes can share a level, in such cases, need to be developed, and are not discussed here.

Note that if $S \leq CL$, and $G > C$, even after temporal partitioning, then it may still be possible to realize the design within a single chip, at the expense of increasing the number of levels.

When we deal with designs that do not fit on a single chip, $S > CL$, we will have to split it across several chips. In such cases, if $G < C$ (for some levelization scheme), then the multi-chip interconnection scheme of Section 8.2.4 can be used to realize the design. If, however, the design is such that $G > C$, in spite of temporal partitioning, then splitting across chips could necessitate duplication of logic.

11.2 Levelization as an integer linear program

This section described how the level allocation problem can be formulated as an integer linear program (ILP). The objective is set as minimizing the maximum of the number of modules per level, G, without violating the fanin and fanout constraints of any node. This objective may seem counter-intuitive, since the constraint imposed by *Dharma* is to have

$$\#(d) \leq C, \ \forall d \tag{11.1}$$

$\#(d)$ is the number of modules in level d. However, if the ILP's solution is $< C$, then the level allocation solution will automatically satisfy 11.1 since $\#(d) \leq G, \forall d$, by the definition of G. If however, the ILP's solution is $> C$, then it implies that it is impossible to satisfy 11.1 without increasing the number of levels.

We consider the level assignment for *slack nodes* only, since the levels of other nodes cannot be changed.

11.2.1 Upper and lower level limits

Let l_i denote the level assigned to node i, using the topological level assignment scheme, starting at the primary inputs. Then, l_i specifies the lowermost level that node i can be assigned to. Similarly, let u_i denote the level assigned to node i, using the topological level assignment scheme, starting at the primary outputs. u_i specifies the uppermost level for node i.

11.2.2 Level variable

We define a variable x_{ij}, to be associated with node i. The j index corresponds to the assignment of node i to level j. x_{ij} is allowed to take only one of two values, 0, 1. If x_{is} takes the value 1, then it means that node i is assigned to level s. Since a node can be assigned to only one level, only one of the l x_{ij}'s can take a value 1. Therefore, for each node i, we have the constraints that

$$\sum_{j=1}^{l} x_{ij} = 1$$

Also, since node i cannot be assigned a level below l_i,

$$x_{ij} = 0, \quad 0 < j < l_i.$$

Similarly, node i cannot be assigned a level above u_i. Therefore,

$$x_{ij} = 0, \quad u_i < j \leq l.$$

Combining the above three equations, we obtain,

$$\sum_{j=l_i}^{u_i} x_{ij} = 1 \tag{11.2}$$

and

$$\sum_{j=1}^{l_i} x_{ij} + \sum_{j=u_i+1}^{l} x_{ij} = 0 \tag{11.3}$$

11.2.3 Level of a node

If $a_l(i)$ denotes the level allocated to node i, then the following equation expresses $a_l(i)$ in terms of x_{ij}.

$$a_l(i) = \sum_{j=1}^{l} j x_{ij} \tag{11.4}$$

This follows directly from the definition of x_{ij}.

11.2.4 Fanin constraints

The circuit specification imposes fanin and fanout constraints. If node i has a fanin k, then the level assigned to node i should always exceed the level assigned to k by at least 1. Similarly, if node i has fanout f, then the level assigned to node f should always exceed the level assigned to i by at least 1.

However, if we specify the fanin constraints for every node, the fanout constraints become redundant, since i's fanout constraint will be covered by f's fanin constraint. Hence, we need to look at only the fanin constraints [2].

So, for node i, with fanin k,

$$a_l(i) \geq a_l(k) + 1$$

Using equation 11.4, and re-arranging the terms,

$$\sum_{j=1}^{l} j(x_{ij} - x_{kj}) \geq 1.$$

Since x_{ij} and x_{kj} are both 0 for $0 < j < l_k$ and $u_i < j \leq l$, because of the upper-limit and lower-limit constraints,

$$\sum_{j=l_k}^{u_i} j(x_{ij} - x_{kj}) \geq 1. \tag{11.5}$$

11.2.5 Nodes per level

At each level j, the number of nodes must be less than G. This is expressed as,

$$\sum_{i=1}^{n} x_{ij} \leq G, \tag{11.6}$$

where n is the number of nodes in the circuit.

11.2.6 Complete ILP

Using equations 11.2, 11.3, 11.5 and 11.6, the complete integer linear program is as follows.

[2] If the fanout node is not a *slack node*, the constraint imposed by it is the same as that imposed by the u_i constraint.

minimize G

Subject to:

$$\sum_{j=l_i}^{u_i} x_{ij} = 1, \quad \forall i$$

$$\sum_{j=1}^{l_i} x_{ij} + \sum_{j=u_i+1}^{l} x_{ij} = 0, \quad \forall i$$

$$\sum_{i=1}^{n} x_{ij} \leq G, \quad \forall j$$

$$\sum_{j=l_k}^{u_i} j(x_{ij} - x_{kj}) \geq 1, \quad \forall k,\ k \in \text{fanin}(i)$$

$$x_{ij} \geq 0$$

$$x_{ij} \leq 1$$

11.3 Temporal partitioning

In the previous section, we showed how to formulate the level allocation problem as an integer linear program. In this section, we present a simple heuristic for the level allocation problem. This heuristic is very fast and effective, and can handle other constraints (such as pass-buffer minimization) in addition to Constraint 11.1. Experimental results are presented in the next section.

11.3.1 Uniform DLM distribution

As before, let there be C DLMs. Let the input circuit (level minimized) have l levels and S modules. Then, the best possible way to distribute the modules is to have at most $\lceil \frac{S}{l} \rceil$ modules per level. If such a module distribution can be obtained, then there will be a maximum utilization of chip resources. It may not be possible to achieve this kind of a distribution for all circuits, because of the constraints imposed by the fanin, fanout and non-slack nodes; however, it is something that is to be aimed for.

We choose this distribution as our objective. That is, given an input circuit, allocate the modules to levels in a manner such that the number of modules in a level i, $\#(i)$, is as close as possible to the number of modules in another level j, for all values of i and j. This is to say that we wish to have a uniform distribution of DLMs across the levels. Mathematically, our objective is to minimize the variance of the DLM distribution.

11.3.2 Modified Kernighan-Lin heuristic

We try to achieve a uniform distribution of DLMs, by using a modified version of the Kernighan-Lin heuristic [Kernighan 70]. Our approach resembles the scheduling techniques used in high-level and data-path synthesis [Park 91], although our constraints and objectives are slightly different.

Kernighan and Lin proposed an algorithm to solve the graph bi-partitioning problem, i.e., given a partition, find two partitions A and B such that the number of edge-crossings are minimum. The Kernighan-Lin (KL) heuristic is based on an iterative improvement scheme. Starting from an arbitrary initial partition A and B, the algorithm seeks to find a group of elements in A to be swapped with a group of elements in B, such that the resulting partitions are optimal. In each iteration of the KL algorithm, a set of movements are selected and carried out, and the next iteration is performed upon the result of the previous iteration. The movements that give the best possible gain improvement (even if negative) in a particular iteration, are the movements that are carried out in that iteration. Allowing negative gain improvements (in other words, an inferior solution) gives the algorithm capability to climb out of local minima. The cumulative gain improvement (sum of the gains in each iteration) is tracked. This gain starts at zero, at the first iteration, and returns to zero at the last iteration (when all elements have changed placed). In between, it will have reached a peak. The sequence of movements up to this peak are retained, and the rest are discarded. The resulting partition is then set as the initial partition, and the entire procedure is repeated until a stage is reached when the initial partition can no longer be improved (this typically takes 2 to 3 iterations in most cases).

In our case, we are performing an l-way partitioning, and the objective is to have uniformly sized partitions. We start from an initial level allocation that satisfies the fanin constraints. Such an allocation can be obtained using topological levelization. Assume we use topological levelization starting from primary outputs. All the slack nodes are identified and formed into an *active set.* Only members of the active set are considered for movement. A move consists of moving a node from its initial level to a new level, without violating fanin or fanout constraints. The cost function of the move is set to minimize the variance of the DLM distribution.

Figure 11.2 outlines our procedure in C-like pseudo-code. Variable ix is the move-index, and is incremented after every move. The variable $\mathcal{G}$ keeps track of the peak cu-

```
repeat{
    ix = 1;
    G = 0; /* peak cumulative gain */
    I = 0; /* index corresponding to peak cumulative gain */
    cum_gain = 0; /* cumulative gain */
    Build an active_set;
    while (active_set not null) {
        Select node i with maximum move_gain;
        Move node i;
        Remove node i from active_set;
        cum_gain = cum_gain + move_gain;
        if cum_gain > G {
            G = cum_gain;
            I = ix;
        }
        Increment ix;
    }
    Only treat the first I moves, from 1 ...I,
    as valid, and undo the rest of the moves.
} until (G ≤ 0)
```

Figure 11.2: Temporal partitioning procedure

mulative gain, and $\mathcal{I}$ stores the move-index corresponding to this gain. To start with, an *active_set* is created. This comprises of the slack nodes. While there are nodes in this set, a node whose move that maximizes the gain is chosen. It is moved, and then removed from the *active_set*. The cumulative gain is incremented by the gain of this move. If the resulting gain surpasses its previous peak value, $\mathcal{G}$ and $\mathcal{I}$ are updated. After all the nodes have been moved, the inner **while** loop terminates. Only the sequence of moves from index 1 to $\mathcal{I}$ are valid, and the other moves are discarded. The procedure within the outer **repeat** loop is repeated until the moves do not cause any further improvements.

11.3.3 Move selection

The criterion for move selection is to minimize the variation of the DLM distribution. Let the current distribution be such that there are d_i DLMs allocated to level i, for $i = 1 \ldots l$. Let us consider the cost of moving DLM m from its current level cur to a new level new. Let the number of modules currently in level cur be c, i.e., $d_{cur} = c$. Similarly, let $d_{new} = n$. If this move is made, then $d_{cur} = c - 1$ and $d_{new} = n + 1$. Let μ represent the mean value of the DLM distribution, i.e., $\mu = S/l$. If v_{cur} and v_{new} represent the current and new variances, then,

$$\begin{aligned} v_{cur} &= \sum_{i=1}^{l} [d_i - \mu]^2 \\ &= [d_{cur} - \mu]^2 + [d_{new} - \mu]^2 + \sum_{i=1, i \neq new, cur}^{l} [d_i - \mu]^2 \\ &= [c - \mu]^2 + [n - \mu]^2 + \sum_{i=1, i \neq new, cur}^{l} [d_i - \mu]^2 \end{aligned} \tag{11.7}$$

Similarly,

$$v_{new} = [(c-1) - \mu]^2 + [(n+1) - \mu]^2 + \sum_{i=1, i \neq new, cur}^{l} [d_i - \mu]^2 \tag{11.8}$$

v_{cur} and v_{new} are non-negative. Hence, if we define $\mathcal{D}_v$ as

$$\mathcal{D}_v = v_{cur} - v_{new}, \tag{11.9}$$

then, $\mathcal{D}_v > 0$ whenever v_{new} is less than v_{cur}. Since we wish to minimize the variance, we choose a move that maximizes $\mathcal{D}_v$. Using equations 11.7 and 11.8 in 11.9,

$$\mathcal{D}_v = [c - \mu]^2 + [n - \mu]^2 - [(c-1) - \mu]^2 - [(n+1) - \mu]^2$$

$$
\begin{aligned}
&= c^2 - 2c\mu + n^2 - 2n\mu - (c-1)^2 + 2(c-1)\mu - (n+1)^2 + 2(n+1)\mu \\
&= c^2 + n^2 - 2\mu[c + n - (c-1) - (n+1)] - c^2 + 2c + 1 - n^2 - 2n - 1 \\
&= 2c - 2n - 2 \\
&= 2(c - n - 1) \\
&= 2(d_{cur} - d_{new} - 1)
\end{aligned}
$$

From this result, we observe that the variance minimizing move is independent of μ, and is dependent only on d_{cur} and d_{new}. Figure 11.3 outlines our move procedure in a C-like pseudo-code. This procedure is called to perform the function of the first line within the inner `while` loop of Figure 11.2. The procedure loops through all nodes in *active_set*. For each node n, the gain is computed using the above formula (the multiply by 2 can be dropped, since we are interested in computing only the maximum, both for the *move_gain* and the *cum_gain* (Figure 11.2)). All the possible levels to which the node can be moved to, are considered, and the gain is computed for each level. Since a node's level has to be greater than all its fanins, levels below $(i_{max}(n)+1)$ are not considered. $i_{max}(n)$ is the level of the fanin of n with the maximum level, i.e.,

$$i_{max}(n) = \max_{f \in fanin(n)} \{a_l(f)\}.$$

Similarly, a node's level has to be lesser than all its fanouts. Hence levels above $(o_{min}(n)-1)$ are not considered, where

$$o_{min}(n) = \min_{f \in fanout(n)} \{a_l(f)\}.$$

The procedure returns the node with the maximum move gain, and its new level position.

11.4 Experimental results

Table 11.1 tabulates the results of using procedures 11.2 and 11.3 on a set of MCNC benchmark examples. The examples were first mapped to 5 input LUTs, with minimum levels, using **mis_pga**[Murgai 91a] [3].

The second column lists the total number of modules, C, in the example, and the third column lists the number of logic levels, l, along the longest path. The $\lceil C/l \rceil$ column is

[3] Any other mapper can be used, like [Cong 92] or **dpmap** of Chapter 5. These experiments were run before **dpmap** was discovered. Since the aim is to study the level distribution, it was felt unnecessary to re-run using other mappers.

```
/* Input is the active_set */
/* Outputs are the node to move, its new level and the move's gain */
Move_select() {
    M_g = 0; /* peak move_gain */
    foreach node n ∈ active_set{
        cur = a_l(n); /* n's current level */
        /* i_max(n) is the max fanin level */
        /* o_min(n) is the min fanout level */
        for (new = (i_max(n) + 1)...(o_max(n) − 1), new ≠ cur) {
            /* consider all possible new levels */
            move_gain = d_cur − d_new − 1;
            if (move_gain > M_g) {
                M_g = move_gain;
                D = n; /* store module to move */
                N = new; /* store new level */
            }
        }
    }
    Return M_g, D and N.
}
```

Figure 11.3: Move selection procedure

to give an idea about the number of modules that would be in each level, if the distribution were perfectly uniform. We start from a topologically levelized graph, levelizing from the primary outputs. The columns under the 'Before' heading correspond to the distribution statistics for this type of levelization. 'Min' is the minimum of the DLMs in any level, and 'Max' is the maximum number of DLMs in any level. σ is the standard deviation of the distribution. The columns under 'After' correspond to similar statistics after temporal partitioning. The last column records the time taken, in CPU seconds, on a DEC 5100, to run the algorithm.

The results are tabulated in ascending order of standard deviation values of the DLM distributions after temporal partitioning. The algorithm is quite successful, and gives phenomenal improvements in several examples (`des`, `rot`, `apex6`, `C5315`). Figure 11.4 shows the results for the `C5315` example, in graphical form.

A few of the examples, however, like `C499` and `alu2`, cannot be partitioned successfully. In such cases, the number of levels may have to be increased to get a more uniform distribution (at the expense of slowing down the circuit), or logic synthesis techniques may be required to synthesize the circuit in a manner that is more suitable for *Dharma* (see Chapter 13).

11.5 Conclusions

In this chapter, we first described an integer linear programming approach to the level allocation problem, and then developed a heuristic for level allocation. The heuristic is called temporal partitioning. Temporal partitioning uses a cost function similar to the Kernighan-Lin heuristic. Experimental results show that the heuristic can yield a fairly uniform DLM distribution for several MCNC benchmark examples.

The move selection procedure (see Figure 11.3) can be modified to take into consideration number of buffers per level, in addition to the number of DLMs per level. Any other types of architectural constraints can also be easily incorporated. This flexibility is the main advantage of the temporal partitioning heuristic.

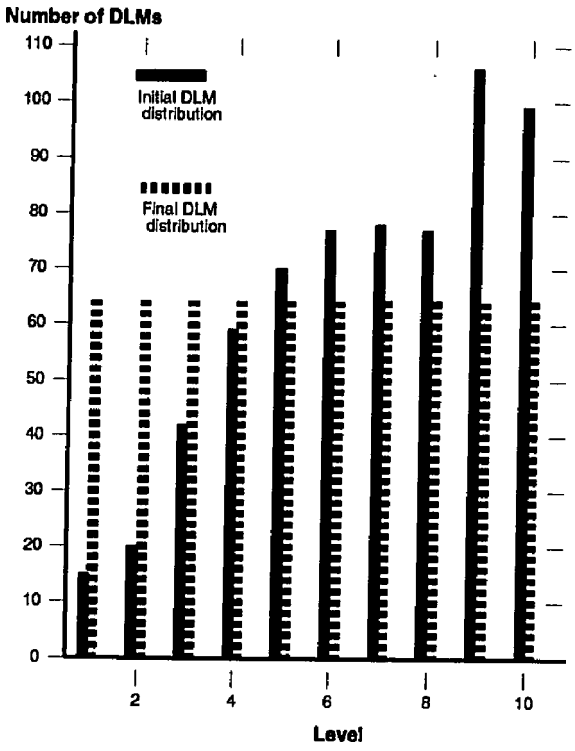


Figure 11.4: DLM distribution before and after temporal partitioning, for the MCNC benchmark example C5315.

Example	Modules C	Levels l	$\lceil C/l \rceil$	Before			After			Time
				Min	Max	σ	Min	Max	σ	CPU s
des	1397	11	127	6	244	69.6	127	127	0.0	135
rot	322	7	46	15	83	22.3	46	46	0.0	4
apex6	274	5	55	9	98	33.5	54	55	0.4	4
apex7	95	4	24	6	36	11.9	23	24	0.4	0
f51m	23	4	5	2	8	2.5	5	6	0.4	0
C5315	643	10	65	15	106	29.0	64	65	0.5	21
misex2	37	3	13	4	18	6.0	12	13	0.5	0
b9	49	3	17	13	18	2.1	15	16	0.5	0
rd84	13	3	5	2	7	2.1	4	5	0.5	0
9sym	7	3	3	1	5	1.9	1	3	0.9	0
duke2	164	6	28	1	48	16.2	25	28	1.1	1
C880	259	9	29	7	41	10.5	30	25	1.4	2
count	81	4	21	5	34	10.9	16	22	2.5	1
sao2	46	5	10	4	15	3.5	4	12	3.0	0
e64	213	5	43	5	67	24.1	40	46	3.0	4
clip	54	4	14	5	21	5.7	5	18	5.1	0
vg2	100	8	13	6	24	6.3	6	20	5.8	0
alu4	155	11	12	3	30	9.0	5	21	6.1	1
apex2	116	6	20	3	33	11.1	3	26	9.0	0
alu2	121	6	21	6	39	13.1	6	31	10.9	1
C499	199	8	25	4	51	14.8	15	42	11.1	1

Table 11.1: DLM distribution statistics before and after temporal partitioning

Chapter 12

Architecture Analysis

Since we are proposing a new architecture, we must analyze the pros and cons, and compare with existing architectures. We perform the analysis under the following sections:

1. Area analysis
2. Timing analysis
3. Experimental verification

12.1 Area analysis

We do a comparison of the area requirements of one particular implementation of the *Dharma* architecture, with existing FPGA devices. We show that the new *Dharma* architecture's area is comparable with the Xilinx and AT&T ORCA devices, and it is therefore feasible to implement *Dharma*.

12.1.1 Assumptions

We proceed with the following assumptions for the *Dharma* chip:

1. DLMs are K-input, 1-output LUTs.
2. There are C DLMs and L maximum levels; assume $K \approx \log_2 L$.
3. There are B buffers; assume $B = 2 \times C$.

4. The DIA is implemented as a one-level crossbar, with two crossbars, the L-crossbar and the B-crossbar. The DLMs are divided into 2 classes, those that connect to even pins, and those that connect to odd pins of DLMs at the next level.

5. There are B input lines, which enter the L-crossbar and B-crossbar through 2-to-1 multiplexers.

6. The chip uses a global level counting scheme as in Figure 8.6.

If the DIA is implemented as a two-level crossbar (like the binomial concentrator) or like a copier (see Section 10.2), it is possible that area requirements will reduce. A sparsely connected K-class DIA will also reduce area requirements. A complex DLM, with interconnected LUTs could also reduce the area (but will increase software complexity).

12.1.2 Area components of *Dharma*

Following are the chief components to be considered for the area analysis:

1. DLM area
   (a) Number of bits ($2^K \times L$ bits per DLM)
   (b) Decoder area (one K-input and one ($log_2 L$)-input decoder per DLM)
   (c) Multiplexer area (one mux per DLM)
   (d) Output buffer area (one buffer per DLM, to drive crossbar)
2. DIA area
   (a) L-crossbar
      i. Number of bits (($B + C$)-input, ($K \times C$)-output crossbar, L bits per crosspoint)
      ii. Number of pass transistors (one per crosspoint)
      iii. Number of lines (one per input and one per output)
   (b) B-crossbar
      i. Number of bits (B-input, B-output crossbar, L bits per crosspoint)
      ii. Number of pass transistors (one per crosspoint)
      iii. Number of lines (one per input and one per output)

iv. Multiplexers (one per primary input and one per DLM output)

3. Latch area (a latch at each DLM and buffer input)

4. Memory configuration circuitry area

5. Buffer area (B buffers to drive crossbar)

12.1.3 The analysis

Configuration bits

			if $B = 2C$
DLM bits	=	$(2^K L) \times C$	$2^K LC$
L-crossbar	=	$\frac{(B+C)(KC)L}{2}$	$\frac{3C^2KL}{2}$
B-crossbar	=	$(B)(B) \times L$	$4C^2L$
PI Mux select bits	=	$B \times L$	$2CL$
Buffer Mux select bits	=	$C \times L$	CL
Total	=	$L\{\frac{KC^2}{2} + (2^K + 1)C + B^2 + B + \frac{KCB}{2}\}$	$LC\{(\frac{3K}{2} + 4)C + (2^K + 3)\}$

Table 12.1: Number of configuration bits

The $\frac{1}{2}$ term in the L-crossbar arises from the odd-even classification of DLMs.

DLM decoders

Assuming an x-y decode scheme, each DLM will have 2 decoders; one K-input decoder to decode the actual DLM inputs, and one $(log_2 L)$-input decoder to decode the level bits. Assuming $K \approx log_2 L$ [1], there are $2C$ K-input decoders.

DLM Muxes

Under the x-y decode scheme, we will require a 2^K-to-1 multiplexer per DLM, hence C such multiplexers are required.

[1] Typically, $K < log_2 L$. So this is a conservative estimate.

Pass transistors

			if $B = 2C$
L-crossbar pass transistors	=	$\frac{(B+C)(KC)}{2}$	$\frac{3C^2K}{2}$
B-crossbar pass transistors	=	$(B)(B)$	$4C^2$
Total	=	$\frac{KCB}{2} + B^2 + \frac{KC^2}{2}$	$(\frac{3K}{2} + 4)C^2$

Table 12.2: Number of pass transistors

Lines

			if $B = 2C$
L-crossbar lines	=	$B + C + KC$	$(3 + K)C$
B-crossbar lines	=	$B + B$	$4C$
Total	=	$3B + (K + 1)C$	(7 + K)C

Table 12.3: Number of routing lines

We assume that each line spans the width or height of the chip (conservative estimate).

2-to-1 multiplexers

C Buffer muxes and B PI muxes are required, giving a total of $(B + C)$ $(= 3C)$ 2-to-1 multiplexers.

Latches

			if $B = 2C$
DLM input latches	=	KC	KC
Buffer input latches	=	B	$2C$
Total	=	$B + KC$	$(K + 2)C$

Table 12.4: Number of latches

Memory configuration circuitry

This circuitry performs the initial configuration of the bits into the configuration bits, in a serial fashion. Hence, the overhead associated with this circuitry can be assumed to be proportional to the number of bits. Let the constant of proportionality be ζ. Hence, area = $LC\zeta\{(\frac{3K}{2}+4)C + 2^K + 3\}$.

Buffer area

$(B + C)$ $(= 3C)$ output buffers to drive crossbar lines.

12.1.4 Comparison with Xilinx 2000 series

The XC2000 has 4-input LUTs and hence the DLMs are directly comparable to these LUTs. Each CLB has 2 outputs (these are different when the LUT is broken into two 3-input LUTs), one latch, three 2-to-1 multiplexers, and three 3-to-1 multiplexers.

The XC2000 has 3 kinds of interconnection resources: general purpose interconnect, long lines, and direct connections.

For the purposes of comparing interconnect resource area, we assume that both the *Dharma* and the XC2000 chip have the same die size. The interconnect lines are measured in terms of chip width and height. Specifically, we assume a square die (width = height), and normalize the interconnection line length to the chip width. In the analysis, it will turn out that *Dharma* will have comparable area to the Xilinx chip (*Dharma* will have smaller area), and hence the assumption of same die size is valid (the assumption is conservative, and favors Xilinx).

The XC2000 has 5 vertical general purpose interconnects [2] and 2 vertical long lines per column; 4 horizontal general purpose interconnects and 1 horizontal long line per row; and 1 direct interconnect in horizontal and vertical direction per CLB (connecting adjacent CLBs). If the chip has P CLBs, there are $5 + 2 + 4 + 1 + 1 + 1 = 14\sqrt{P}$ chip-width lines.

Each line can be connected to the CLB inputs/outputs and other lines by means of programmable interconnect points (PIPs). The Xilinx data book does not specify the number of pass-transistors, but as a lower bound, we assume that each PIP has one pass-

[2] Each general purpose interconnect line is segmented, segments being connected by means of switch matrices. One column(row) of such segments ($\sqrt{P}$ segments) is treated as one chip-width line in this comparison.

Area component	XC2064	*Dharma*	Remarks
Number of bits	11254	$8 \times 8\{(\frac{3\times4}{2}+4)8+2^4+3\}$ $= 6336$	Xilinx data book
Decoders	64, 4-input	16	8 (of 16) are 3-input for *Dharma*
DLM Mux	None	8, 16-to-1	(mux + decoder) area for *Dharma* < XC2064 decoder area
Pass transistors	74×64 $= 4736$	$(\frac{3\times4}{2}+4)8^2$ $= 640$	
Lines	14×8 $= 112$	$(7+4)8$ $= 88$	
2-to-1 mux	3×64 $= 192$	3×8 $= 24$	
3-to-1 mux	192	None	
Latches	64	$(4+2)8$ $= 48$	
Memory config	11254ζ	6336ζ	same overhead / bit
Buffer	2×64 $= 128$	3×8 $= 32$	

Table 12.5: *Dharma*'s area comparison with XC2064

transistor [3]. From the XACT software's plot of the Xilinx chip, we count 50 PIPs surrounding each CLB. There are 2 switch matrices per CLB, and each switch matrix is 2×2. Assuming 2 switch points per matrix, with 6 pass transistors per point, we have 24 pass-transistors per CLB, in the switch matrix. Thus we have a total of $74P$ pass-transistors. We assume 2 output buffers per CLB, resulting in $2P$ buffers.

XC2064

The XC2064 has 64 CLBs, therefore $P = 64$. We compare it with a *Dharma* chip with $C = 8; K = 4; L = 8$ (note: $K < \log_2 L$).

Table 12.5 shows the comparison. The *Dharma* resources use less area. It is reasonable to assume that the DLM mux and decoder areas for *Dharma* will sum up to be less than the decoder areas of the XC2064.

[3]This may be a poor lower bound, since PIPs may have more than one pass transistor to allow connections in different directions. However, we choose to err on the safer side.

Area component	XC2018	*Dharma*	Remarks
Number of bits	17238	$10 \times 10\{(\frac{3\times4}{2}+4)10+2^4+3\}$ $= 11900.$	Xilinx data book
Decoders	100, 4-input	16, 4-input	
DLM Mux	None	10, 16-to-1	(mux + decoder) area for *Dharma* < XC2018 decoder area
Pass transistors	74×100 $= 7400$	$(\frac{3\times4}{2}+4)10^2$ $= 1000$	
Lines	14×10 $= 140$	$(7+4)10$ $= 110$	
2-to-1 mux	3×100 $= 300$	3×10 $= 30$	
3-to-1 mux	300	None	
Latches	100	$(4+2)10$ $= 60$	
Memory config	17238ζ	11900ζ	same overhead / bit
Buffer	2×100 $= 200$	3×10 $= 30$	

Table 12.6: *Dharma*'s area comparison with XC2018

XC2018

The XC2018 has 100 CLBs, therefore $P = 100$. We compare it with a *Dharma* chip with $C = 10; K = 4; L = 10$ (note: $CL = P$ and $K < \log_2 L$). Table 12.6 shows the comparison. It is reasonable to assume that the DLM Mux and decoder areas for *Dharma* will sum up to be less than the decoder areas of the XC2018. Again, the *Dharma* resources use less total area.

12.1.5 Comparison with Xilinx 3000 series

The Xilinx 3000 series chips use an interconnect pattern similar to the 2000 series. The CLBs are different, however. Each CLB has 2 latches, one 5-input LUT that can be configured to implement two 4-input functions, seven 2-to-1 muxes, two 3-to-1 muxes and one 2-input OR gate.

As in the 2000 series, we assume that XC3000 chip and *Dharma* have similar die sizes, and normalize line lengths to chip-width. There are 5 general purpose vertical lines and 3 vertical long lines per column, and 5 general purpose horizontal lines and one horizontal

Area component	XC3090	*Dharma*	Remarks
Number of bits	62668	$16 \times 20\{(\frac{3\times5}{2}+4)20+2^5+3\}$ $= 84800.$	Xilinx data book
Decoders	320, 5-input	40, 5-input	
DLM Mux	None	20, 32-to-1	
Pass transistors	100×320 $= 32000$	$(\frac{3\times5}{2}+4)20^2$ $= 4600$	
Lines	$9 \times 20 + 7 \times 16$ $= 292$	$(7+5)20$ $= 240$	
2-to-1 mux	7×320 $= 2240$	3×20 $= 60$	
3-to-1 mux	2×320 $= 640$	None	
Latches	640	$(5+2) \times 20$ $= 140$	
Memory config	62668ζ	84800ζ	same overhead / bit
Buffer	2×320 $= 640$	3×20 $= 60$	

Table 12.7: *Dharma*'s area comparison with XC3090

long line per column. A direct interconnect from a CLB output connects to adjacent CLBs in the horizontal and vertical directions. Hence there are a total of $(5+3+1)r+(5+1+1)c = 9r + 7c$ chip-width lines, where r is the number of rows and c is the number of columns.

Each line can be connected to the CLB inputs/outputs and other lines by means of PIPs. We use the number of PIPs as the lower bound on the number of pass-transistors, although each PIP may have more than one pass transistor to implement connections in various directions. From the data book and the XACT software's detailed diagram of the XC3000 series, we calculated the number of pass-transistors to be $100P$, where $30P$ pass-transistors are used in the 5×5 switch matrix (5 switch points, with 6 pass transistors per point), and $70P$ PIPs surround CLBs. There are $2P$ output buffers to drive the lines.

XC3090

The XC3090 has 320 CLBs, each with a 5-input LUT. Therefore, $P = 320$. There are 20 rows and 16 columns, which implies $r = 20$ and $c = 16$. We use *Dharma* parameters as $C = 20; K = 5; L = 16$ (note: $CL = P$ and $K < \log_2 L$).

Table 12.7 shows the comparison. Since it is difficult to make a comparison directly,

Component	XC3090	*Dharma*
Bits	62668	84800
Decoders	320 x 64 = 20480	40 x 64 = 2560
DLM mux	0	20 x 32 = 640
Pass transistor	16000	2300
2-to-1 mux	2240	60
3-to-1 mux	640 x 4 = 2560	0
Latches	640 x 3 = 1920	140 x 3 = 420
Memory config	62668	84800
Buffer	640 x 2 = 1280	60 x 2 = 120
Total	169816	175700

Table 12.8: Gate equivalent area comparison with XC3090

Area component	XC3090	*Dharma*
Number of bits	62668	$16 \times 20\{(\frac{3\times5}{5}+4)20+2^5+3\} = 56000.$
Pass transistors	32000	$(\frac{3\times5}{5}+4)20^2 = 2800$

Table 12.9: K-class DIA fares better

we use gate equivalent analysis. We assume that each bit is equivalent to 1 gate, each DLM mux is equivalent to 32 gates (wired OR, uses column select from column decoder), each decoder is equivalent to 64 gates, each latch is 3 gate equivalents, each memory configuration overhead is 1 gate equivalent per bit, each buffer is 2 gate equivalent, each 2-to-1 mux is 1 gate-equivalents, each 3-to-1 mux is 4 gate-equivalents, and each pass transistor is 0.5 gate equivalent. We then have the gate-equivalent totals as in Table 12.8.

Thus, the *Dharma* chip is bigger by a factor of 3.5%.

If we use a K-class division, for the DLM connections to the next level DLM inputs (i.e., each DLM output connects to only one input of the next level DLM), the number of bits and pass transistors changes as tabulated in Table 12.9.

Therefore, the gate equivalents decrease by $2(84800 - 56000) + 0.5(4600 - 2800) =$

Size	XC4000	*Dharma*
64	35956	$C = 8; L = 8; K = 5$ $8 \times 8\{(\frac{3\times5}{2} + 4)8 + 2^5 + 3\}$ $= 8128$
100	51788	$C = 10; L = 10; K = 5$ $10 \times 10\{(\frac{3\times5}{2} + 4)10 + 2^5 + 3\}$ $= 15000$
196	92092	$C = 20; L = 10; K = 5$ $10 \times 20\{(\frac{3\times5}{2} + 4)20 + 2^5 + 3\}$ $= 53000$
324	143916	$C = 20; L = 16; K = 5$ $16 \times 20\{(\frac{3\times5}{2} + 4)20 + 2^5 + 3\}$ $= 84800$
400	174148	$C = 25; L = 16; K = 5$ $16 \times 25\{(\frac{3\times5}{2} + 4)25 + 2^5 + 3\}$ $= 12900$

Table 12.10: *Dharma*'s configuration bits comparison with XC4000

58500, and becomes 117200, 31% less than the XC3090.

12.1.6 Comparison with Xilinx 4000 series

A direct one-to-one comparison with the XC4000 series is not possible because of the significant difference in the CLB structure. We cannot compare one CLB with one DLM, unless we change the DLM structure itself. However, from the analysis so far, it is clear that the dominating factor is the number of bits, and we hence tabulate, in Table 12.10, the number of bits for the XC4000 series chips and the *Dharma* chips (odd-even class interconnection) with $CL = P$, where P is the number of CLBs in the XC4000 series chips. We assume a 5-input LUT for the DLM.

From the table, we observe that *Dharma* uses far fewer configuration bits.

12.1.7 Comparison with the AT&T ORCA

The comments on comparison with the XC4000 apply to the AT&T ORCA chips as well. We just list the number of bits in Table 12.11. Again, *Dharma* uses far fewer configuration bits.

Size	ORCA	*Dharma*
100	42816	$C = 10; L = 10; K = 5$ $10 \times 10\{(\frac{3\times5}{2} + 4)10 + 2^5 + 3\}$ $= 15000$
196	81048	$C = 20; L = 10; K = 5$ $10 \times 20\{(\frac{3\times5}{2} + 4)20 + 2^5 + 3\}$ $= 53000$

Table 12.11: *Dharma*'s configuration bits comparison with AT&T ORCA

12.2 Timing analysis

In this section, timing equations for *Dharma* are derived. The equations are dependent on the clocking scheme used, and hence there are two equations, *EFP timing equation* and *S timing equation*. Each equation gives the total time required for signals to propagate from the inputs to the outputs, for an l level circuit, when the circuit is implemented on a *Dharma* chip.

12.2.1 Notation

The following lists the symbols used in the timing equations.

D_{EFP}	Delay from input to output for EFP clocking.
D_S	Delay from input to output for S clocking.
f_E	*Ext clk* frequency.
f_I	*Int clk* frequency.
l	Number of levels in the circuit being implemented.
T_d	Clock period of *Int clk*.
T_h	Time duration for which *Int clk* is high.
T_l	Time duration for which *Int clk* is low.
t_{dlm}	Worst-case propagation time through DLM.
t_{dia}	Worst-case propagation time through DIA.
t_{Ircfg}	Reconfiguration time for DIA.
t_{Lrcfg}	Reconfiguration time for DLM (= 0 for LUT DLM).
t_{ld}	Latch/Register propagation time.
t_{lh}	Latch/Register hold time.
t_{ls}	Latch/Register set-up time.
t_{mux}	Worst-case propagation time through a 2-input multiplexer.
t_{xbL}	Worst-case propagation time through the L crossbar.
t_{xbB}	Worst-case propagation time through the B crossbar.

12.2.2 EFP timing equation

Consider Figures 8.7(a), 8.13 and 12.1.

In Figure 12.1, the path from input register to output register has been unfolded. The unfolded path has been drawn in a snake like fashion. All possible connections (DLM to DLM, DLM to buffer, buffer to buffer, and buffer to DLM) have been shown in the figure. The vertical dotted line, labeled **H** corresponds to the time instant at which *Int clk* goes high, and the vertical dotted line labeled **L** corresponds to the time instant at which *Int clk* goes low. The various components have been drawn such that they are aligned to these two time instances. The jog in the vertical line **L** corresponds to the extension in T_l, the low portion of *Int clk*, to account for the delay through the primary input registers.

In Figure 8.13, the reconfiguration time of the DIA is time-shared with the propagation time through the DLM, and this occurs when *Int clk* is high. The DIA can be reconfigured after a delay of t_{lh}, after latching the inputs, for proper latch operation.

From Figure 12.1 and 8.13,

$$\begin{aligned} T_h &\geq t_{Ircfg} + t_{lh} \\ T_h &\geq t_{dlm} + t_{ld} \end{aligned}$$

Combining these two equations,

$$T_h \geq \max\{(t_{Ircfg} + t_{lh}), (t_{dlm} + t_{ld})\} \tag{12.1}$$

From Figure 12.1, the worst-case delay through the DIA is given by

$$t_{dia} = \max\{(t_{mux} + t_{xbL}), (2 \times t_{mux} + t_{xbB})\} \tag{12.2}$$

Since the B crossbar is much smaller than the L crossbar, it is likely that $t_{dia} = t_{mux} + t_{xbL}$.

After signals have propagated through the interconnect, an additional duration of t_{ls} is required before *Int clk* can go high. Therefore,

$$T_l \geq t_{dia} + t_{ls} \tag{12.3}$$

For the first period, however, *Int clk* has to remain low for an extra duration of t_{ld} (propagation through the primary input register).

Combining equations 12.1 and 12.3,

$$\begin{aligned} T_d &= T_h + T_l \\ &\geq \max\{(t_{Ircfg} + t_{lh}), (t_{dlm} + t_{ld})\} + t_{dia} + t_{ls} \end{aligned} \tag{12.4}$$

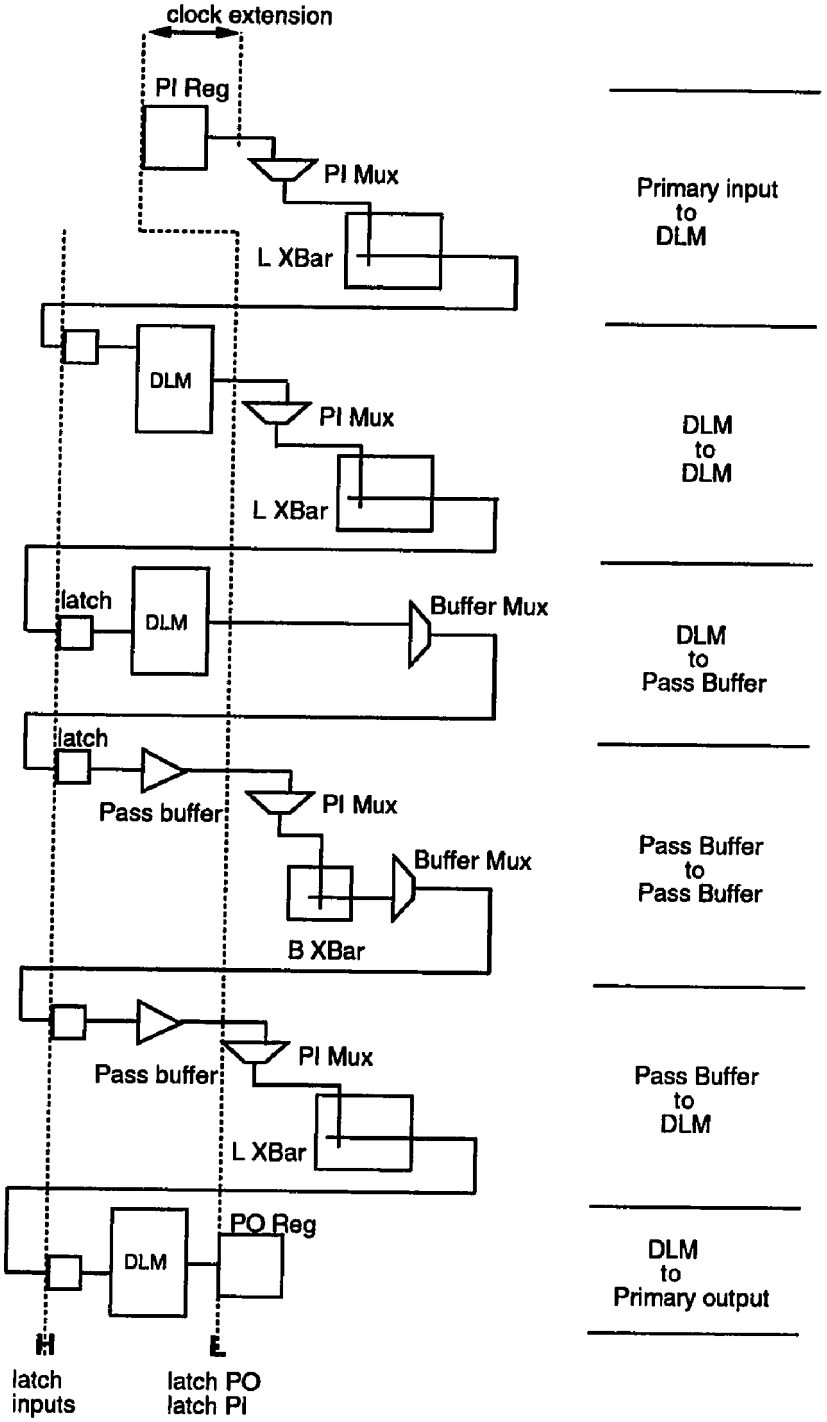


Figure 12.1: Input to output signal path for EFP clocking scheme

For an l level circuit, there are l durations of T_h, and l durations of T_l, in the EFP scheme. The first T_l is extended by t_{ld}. Hence, the delay from input to output,

$$\begin{aligned} D_{EFP} &= l \times T_h + l \times T_l + t_{ld} \\ &= l \times T_d + t_{ld} \end{aligned} \tag{12.5}$$

Substituting equation 12.4 in 12.5, and assuming that *Int clk* operates at the maximum rate,

$$D_{EFP} = l \times [\max\{(t_{Ircfg} + t_{lh}), (t_{dlm} + t_{ld})\} + t_{dia} + t_{ls}] + t_{ld} \tag{12.6}$$

Equation 12.6 is the EFP timing equation. The delay through the implemented circuit varies linearly as the number of logic levels, l. This is a key feature of the *Dharma* architecture.

If $(t_{Ircfg} + t_{lh}) < (t_{dlm} + t_{ld})$ (as will usually be the case, especially if the DLM is an interconnection of LUTs), then

$$D_{EFP} = l \times [t_{dlm} + t_{dia} + t_{ld} + t_{ls}] + t_{ld} \tag{12.7}$$

Equation 12.5 gives the minimum clock period of *Ext clk*. If f_I is *Int clk* frequency, and f_E is the *Ext clk* frequency,

$$\begin{aligned} f_E &= \frac{1}{D_{EFP}} \\ &= \frac{1}{l \times T_d + t_{ld}} \\ &= \frac{1}{T_d(l + \delta)} \\ f_E &= \frac{f_I}{l + \delta} \end{aligned} \tag{12.8}$$

where $\delta = \frac{t_{ld}}{T_d}$. Thus, *Ext clk* is slower than the *Int clk* frequency by a factor of $\frac{1}{l+\delta}$ for EFP clocking.

12.2.3 S timing equation

Figure 12.2 shows the unfolded signal path for the S clocking scheme. Comparing with Figure 12.1, in Figure 12.2 the PI and PO registers are clocked in phase with the input latches, and hence are placed vertically aligned to the **H** dotted line. As a result of this shift

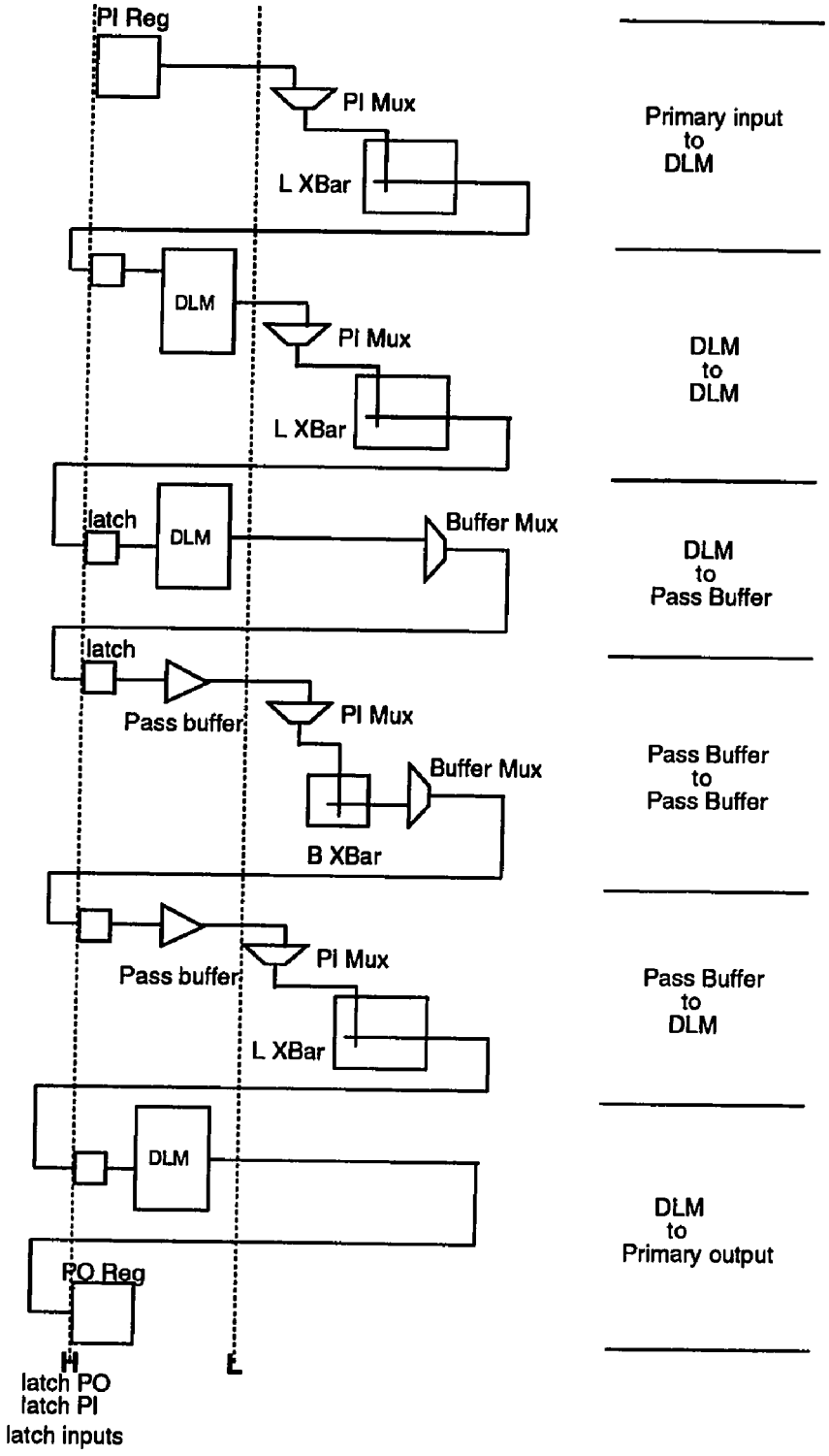


Figure 12.2: Input to output signal path for S clocking scheme

in time, the propagation time through the primary input register (t_{ld}) is now in phase with the propagation time through the input latches, and hence does not cause an extension of the clock period. However, this phase alignment increases the number of *Int clk* cycles to $(l+1)$, as shown in Figure 8.7(b). The rest of Figure 12.2 is similar to Figure 12.1. The derivation of T_d remains unchanged, and is exactly as given in equation 12.4. The S timing equation, for an l level circuit implemented on *Dharma* is given by

$$D_S = (l+1) \times T_d \tag{12.9}$$

Substituting equation 12.4,

$$D_S = (l+1) \times [\max\{(t_{Ircfg} + t_{lh}), (t_{dlm} + t_{ld})\} + t_{dia} + t_{ls}] \tag{12.10}$$

If $(t_{Ircfg} + t_{lh}) < (t_{dlm} + t_{ld})$, then

$$D_S = (l+1) \times [t_{dlm} + t_{dia} + t_{ld} + t_{ls}] \tag{12.11}$$

From equation 12.9,

$$f_E = \frac{f_I}{l+1} \tag{12.12}$$

So, *Ext clk* is $\frac{1}{l+1}$ times slower than *Int clk* in the S clocking scheme.

Comparing equation 12.11 with 12.7, the circuit will be slower by $t_{dlm} + t_{dia} + t_{ls}$. Hence the "slow" prefix for this clocking scheme.

12.2.4 Remarks

The time required by a K-input LUT CLB for logic function evaluation will be less than the time required by a DLM of equal functional capability, since the DLM is essentially a $(K + log_2 L)$-input LUT. However, since there will be fewer decoders (by a factor of L), they can be made larger to provide the same speed as that of a K-input LUT CLB.

The interconnect delay (see equation 12.2) is a constant predictable delay, whose value depends on the crossbar implementation (several types are possible, as described in Section 10.2). Typical values of 4 to 5 ns delay for t_{dia} is expected [Horng 92]. The sum of the latch set-up and propagation times is of the order of 3 to 4 ns. Thus, the latch and interconnect together introduce a latency of about 7 to 9 ns. While the Xilinx style architecture boasts of a much smaller interconnection delay, experiments over actual circuits show that routing delay could be as high as 48 ns.

12.3 Experimental verification

We performed a comparison of *Dharma*'s timing behavior with that of the Xilinx 3000 series chips. Our experiment is conducted in the following manner: Several MCNC combinational benchmark circuits were mapped to Xilinx using the **mis_pga** mapping technique [Murgai 91a]. The mapper generates an output circuit with 1-output LUTs, each having ≤ 5 inputs. One such LUT is assigned to each CLB on the Xilinx chip. Each benchmark circuit is placed and routed (using the Xilinx place and route tool **apr**) on the smallest possible XC3000 series chip (i.e., the smallest chip which satisfied the condition, # CLBs $\geq$ # LUTs). However, if there were unrouted nets on such a chip, the next larger chip is used.

The results are tabulated in Table 12.12 (see Table 7.1 also). The second column lists the number of 1-output LUTs in the examples, and the third column lists the number of levels of logic on the path with maximum delay. The 'Xilinx delay' column lists the worst-case delay as computed by Xilinx's timing analyzer **XDelay**.

The temporal partitioning technique of Section 11.3 is used to levelize the LUTs for *Dharma*. Columns 'Max DLMs' and 'Max buffers' list the statistics after temporal partitioning. 'Max DLMs' is the maximum number of DLMs in any level, and 'Max buffers' is the maximum number of buffers in any level. A *Dharma* chip with appropriate C and B is selected to implement the circuit, based on these numbers. The delays for the *Dharma* implementation are calculated using equations 12.7 and 12.11 for the EFP and S clocking schemes respectively, and these are listed in the columns towards the right hand side of Table 12.12. The Xilinx delay numbers include IOB delays; hence additional IOB delays are added to the *Dharma* delay numbers as well. The percentage decrease in *Dharma* delays, as compared to the Xilinx delays are listed in the '% impr.' columns.

In computing the expected delay on *Dharma*, we make the following assumptions.

1. Xilinx CLB delay = *Dharma* DLM delay (t_{dlm}) = 8 ns.
2. Xilinx IOB delay = *Dharma* IOB delay = 6 + 27 = 33 ns.
3. *Dharma* interconnect delay (t_{dia}) = 5 ns
4. *Dharma* latch set-up delay (t_{ls}) = 0.2 ns.
5. *Dharma* latch propagation delay (t_{ld}) = 3.8 ns.

Example	LUTs	Levels	Max DLMs	Max buffers	Xilinx delay	*Dharma*			
						EFP clock		S clock	
						delay	% impr.	delay	% impr.
5xp1	21	2	11	7	87.8	71.0	-19.1	84.0	-4.3
C499	199	8	42	59	346.3	173.0	-50.0	186.0	-46.3
C880	259	9	30	110	342.9	190.0	-44.6	203.0	-40.8
alu2	121	6	31	24	187.4	139.0	-25.8	152.0	-18.9
alu4	155	11	21	42	316.5	224.0	-29.2	237.0	-25.1
apex7	96	4	24	53	168.6	105.0	-37.7	118.0	-30.0
b9	49	3	16	38	107.2	88.0	-17.9	101.0	-5.8
bw	28	1	28	0	75.6	54.0	-28.6	67.0	-11.4
clip	54	4	18	18	120.5	105.0	-12.9	118.0	-2.1
count	81	4	22	43	133.1	105.0	-21.1	118.0	-11.3
duke2	164	6	28	72	195.2	139.0	-28.8	152.0	-22.1
e64	213	5	46	90	203.5	122.0	-40.0	135.0	-33.7
f51m	23	4	6	13	109.3	105.0	-3.9	118.0	8.0
misex2	37	3	13	24	93.3	88.0	-5.7	101.0	8.3
sao2	46	5	12	14	125.4	122.0	-2.7	135.0	7.7
vg2	100	8	20	17	151.9	173.0	13.9	186.0	22.4
Average							-22.1		-12.8

Table 12.12: Table showing results of an experiment comparing *Dharma* with Xilinx's 3000 series. Xilinx results were obtained after place and route, using **apr**, followed by **Xdelay**'s timing analysis.*Dharma* results are from the timing equations 12.7 and 12.11.

From these estimated delays for *Dharma*, we observe an average improvement of 22.1% if the EFP clocking scheme is used, and an improvement of 12.8% if the S clocking scheme is used, for the 16 MCNC benchmark circuits.

Since faster DIAs are possible, there is a further scope for improving the *Dharma* delay numbers. Also, the mapping for *Dharma* has only been done using temporal partitioning; i.e, no real logic synthesis has been performed (see Chapter 13 for a possible approach for logic synthesis for *Dharma*). Logic synthesis geared for *Dharma* can further reduce the timing delay, and also reduce resource utilization (number of DLMs and buffers used).

12.4 Conclusions

In this chapter, the new *Dharma* architecture's silicon area usage and timing behavior were studied. We compared *Dharma*'s area usage with existing reconfigurable architectures, such as the Xilinx series of chips and the AT&T ORCA. We derived timing equations for *Dharma*, which provide timing delay and clock rates for circuits implemented on *Dharma*.

Our area analysis showed that the silicon area of the *Dharma* device is expected to be smaller compared to the XC 2000 series chips, for the same logic capability. *Dharma* has about the same area as the XC 3090 and *Dharma* has far fewer configurable memory bits (and switches) compared to the XC 4000 and AT&T ORCA FPGAs. Timing performance estimates of *Dharma* compared to the XC 3000 series show that circuits implemented on *Dharma* can be faster by about 13% if the S clocking scheme is used, and by about 22% if the EFP clocking scheme is used.

One may tend to argue that the *Dharma* numbers show only a few percent improvement, and it may therefore not be worth the effort to go for a *Dharma* style FPLD. However, it must be pointed out that in our analysis, we have taken a conservative approach in favor of Xilinx. It was necessary to be conservative, since our approach is based on estimates (a *Dharma* device has not yet been implemented). Estimates can sometimes be off by a few percent, and it is best to err on the safer side; so that errors in the estimate, if any, can only be to *Dharma*'s advantage. Thus, our analysis yields the worst-possible numbers for *Dharma*. Synthesis results are sub-optimal; synthesis tailored for *Dharma* can improve timing behavior. We have assumed an SRAM technology – since this is the one used by Xilinx and AT&T. However, as discussed in Chapter 10, the *Dharma* architec-

ture is amenable to other technologies such as DRAM and CCD, which can result in area savings and better timing performance (Xilinx and ORCA devices cannot readily benefit from these other technologies). DIA size can be reduced by using the full K-class DIA (see Section 10.2.6), and interconnect delay can be reduced by using the precharge and sense scheme (see Section 10.2.5).

In spite of the conservative approach, *Dharma* has better silicon area and timing performance. With a suitable choice of technology and architectural modifications (from amongst the several listed in Chapter 10), and with a set of CAD tools specifically developed for *Dharma*, it is expected that a *Dharma* based PLS will have clear-cut advantages.

Chapter 13

Unexplored Terrain

The purpose of this chapter is to point out potential avenues for furthering the work on the *Dharma* architecture. One is to build a prototype device. This may be straightforward for expert designers, and is not discussed here. Chapter 11 only discussed synthesis as a level allocation problem. In the first section here, we describe the logic synthesis requirements for *Dharma*. Next, we look at improvements possible in the overall architecture, if software techniques are available to solve certain special problems that arise. Our objective is overall PLS improvement, and architecture improvements that cannot be exploited by the accompanying software are not real improvements. We describe a synthesis problem which, if solved, can cause an improvement in the area and performance of the device.

13.1 Logic synthesis for *Dharma*

From a logic synthesis viewpoint, *Dharma* poses a unique problem. Given a circuit specification, the synthesizer should generate a network with the following objectives.

1. Number of levels must be minimized.
2. Number of modules in any level d, $M_d = \max(\#(d))$, must be minimized.
3. Number of inter-level signals (signals that travel more than one level) must be minimized.

Minimizing the number of levels improves the performance since the delay through the *Dharma* realization is proportional to l, the number of circuit levels. Minimizing l also

minimizes the number of configuration bits, since these are also proportional to l (i.e., by minimizing l, one can use a smaller *Dharma* chip).

The circuit is implemented on a *Dharma* device which has $C \geq M_d$. Thus, minimizing M_d allows one to use a smaller chip. Similarly, the number of inter-level signals determines the number of on-chip pass-buffers required, and again minimizing this number, can allow a smaller chip to be used for the circuit realization.

Our approach in Chapter 11 partitioned the above objectives into two. Objective 1 was solved with no regard to the other two constraints. However, we conjecture that a logic synthesizer that considers all the three objectives together can give better results compared to those reported in Chapter 11.

One possible approach to this synthesis problem, is along the lines of [Kim 93]. Although the approach used in [Kim 93] is for wave-pipelined circuits, the objectives are somewhat similar. In the case of wave-pipelining, the logic synthesis solution should be such that the resulting circuit is *balanced.* That is, the shortest and longest paths at every node, from the primary inputs, must be equal. One way of achieving this balance is to add delay buffers to the shorter path so that it becomes as slow as the longer path. However, this is not the best way, as the simple case in Figure 13.1 (taken from [Kim 93]) illustrates. Figure 13.1(a) is one possible implementation of the function

$$y = ((a + b)\, c + d)\, e + d\, g.$$

To balance this circuit requires 10 delay buffers. However, the same function can be realized as in Figure 13.1(b). The second realization is balanced with only one delay buffer. The authors in [Kim 93] present a *balancing algorithm* which restructures a given circuit into a balanced circuit, keeping the area (area due to gates plus delay buffers) and delay minimum.

The balancing algorithm first decomposes the input circuit into a network of 2-input gates. Then the algorithm starts at the most unbalanced (largest difference between longest and shortest paths) output node. This node is collapsed, with a limit on the size of the collapsed node. Unbalanced fanin nodes are recursively balanced. The collapsing is followed by a balanced decomposition.

The balanced decomposition is the key step in the balancing algorithm (and it is this step that needs to be modified for *Dharma*). For a balanced decomposition, the kernel and co-kernel nodes must have equal logic depths, which is one less than the remainder node's logic depth. The cost function to choose a kernel for decomposition is set using

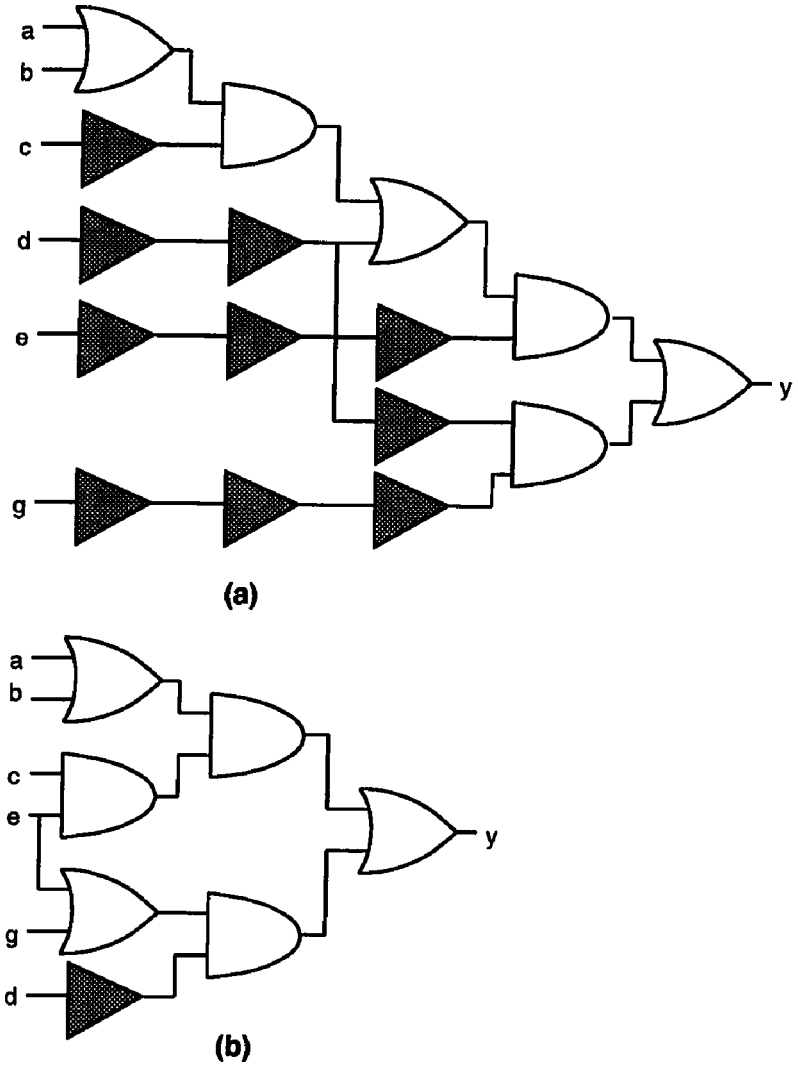


Figure 13.1: Unbalanced and balanced implementation of function y.

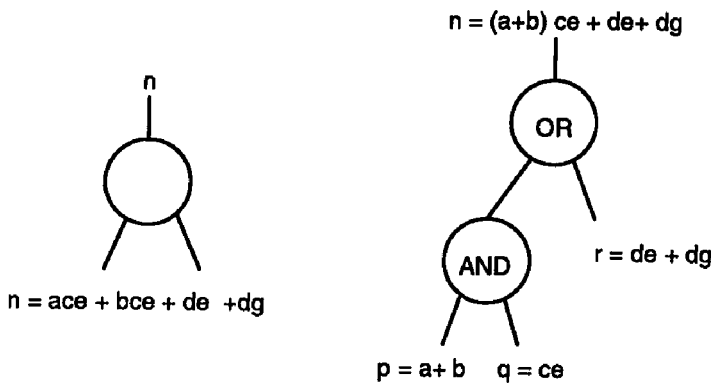


Figure 13.2: Example of balanced decomposition.

this requirement. In Figure 13.2, function n is being decomposed. A candidate kernel is $p = (a + b)$. The corresponding co-kernel is $q = c\ e$ and the remainder is $r = d\ e + d\ g$. Hence, if p and q have the same level (i.e., logic depth), and r is at the next level, then this decomposition results in balancing node n.

Therefore, the quality of a kernel p for the decomposition of node n is chosen as

$$\begin{aligned} cost(n,p) &= \| delay(p) - delay(q) \| + \\ &\quad \| delay(r) - MAX(delay(p), delay(q)) - gate\_delay \| \end{aligned}$$

A kernel with the least cost is chosen for the decomposition.

For *Dharma*, balancing is relevant since it minimizes pass-buffer usage. In addition to balancing, in *Dharma*, the number of nodes per level must be minimized. This must be incorporated into the kernel selection cost function. One way to do this could be to compute

$$l\_cost(n,p) = \#(level(p)) + \#(level(q)) + \#(level(r),$$

where $\#(d)$ is the number of nodes at level d. l_cost could then be added to $cost(n,p)$ with a weighting factor. The estimation of delay is straightforward in the case of *Dharma*, because of the linear relationship to the logic level (this is not the case for standard cells and other design styles, for which the above algorithm was intended, and the authors describe heuristics to estimate delay in such cases). The number of inputs per node, and the type of kernel selection will also have to be modified for *Dharma*, based on the logic function

capabilities of the DLMs. For example, if the DLM is a K-input LUT, then LUT based decomposition strategies can be incorporated.

13.2 Repetitive structures

In Chapter 10, we discussed the sparsely connected K-class DIA. This technique reduces the number of cross-points by almost a factor of K. However, it necessitates a graph-coloring type of placement algorithm to perform the placement of the modules in such a manner that there are no unrouted nets.

Here we discuss a very sparse crossbar (VSC) DIA. Such a crossbar requires sophisticated synthesis techniques to make sure that the given circuit can be fully routed. Unless such techniques are developed, the VSC cannot be used. In this section, we motivate the principle behind the VSC, and outline synthesis requirements for the same.

13.2.1 Principle

If the combinational part of a circuit can be synthesized in a manner such that there is a repetition of interconnection patterns, then part of the reconfigurable crossbar can be made of *fixed* connections instead of programmable connections. Let $\mathcal{P}_i$ be a matrix of 0's and 1's such that the rows of $\mathcal{P}_k$ correspond to the horizontal lines of the DIA and the columns to the vertical lines. An element $a_{ij} = 1$ implies that vertical line j connects to horizontal line i at level k, and a '0' implies that there is no connection. If we can synthesize a mapping such that the $\mathcal{P}_i$'s are as identical as possible, i.e.,

$$\| \mathcal{P}_2 - \mathcal{P}_1 \| \approx 0, \| \mathcal{P}_3 - \mathcal{P}_2 \| \approx 0, \| \mathcal{P}_4 - \mathcal{P}_3 \| \approx 0, \cdots,$$

then it means that certain configuration bits in the interconnection remain the same every level, and can hence be fixed instead of programmable. Which portion is to be kept fixed is dependent on the ability of the synthesis algorithm.

13.2.2 Example

As an example, consider the tiny *Dharma* architecture of Figure 13.3. Each DLM is a 2-input LUT, with 1 output. The L-crossbar has only 8 programmable points. There are four fixed connections. Let us assume that there are no buffers, for the purposes of this

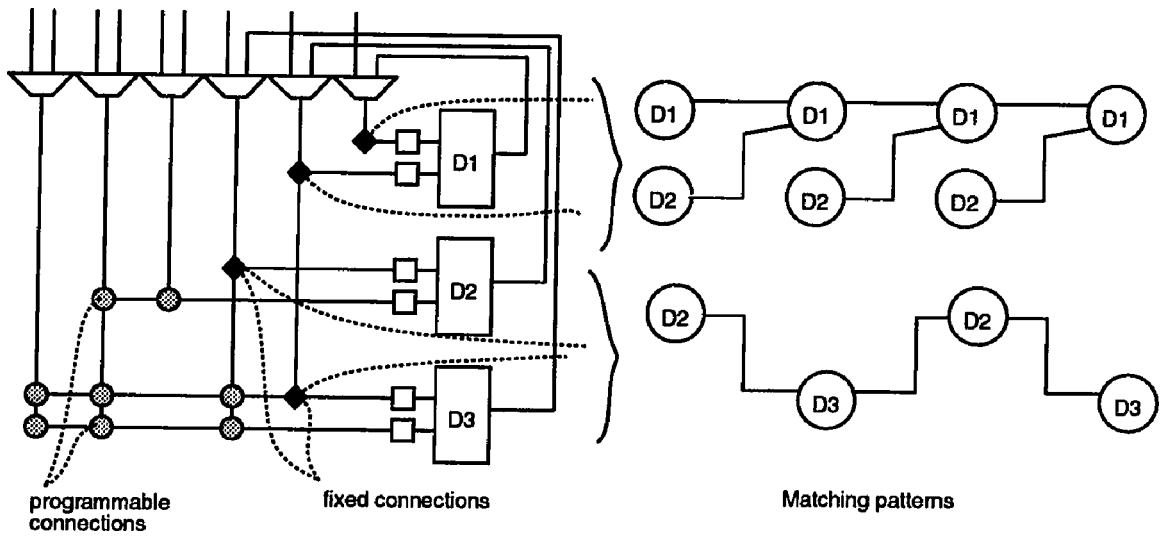


Figure 13.3: Tiny *Dharma* with VSC.

illustration. DLMs *D1* and *D2* are connected to DLM *D1* through fixed connection points (through the PI muxes). These connections can implement a portion of the circuit that has the repetitive pattern shown on the right side of Figure 13.3.

Similarly, DLM *D3* is connected to *D2*, and vice-versa, using fixed connections. Portions of the circuit that match the pattern on the right hand side of Figure 13.3 can be mapped on to *D2* and *D3*, using these connections.

The programmable connections allow other portions of the circuit to be realized (we cannot expect every part of the circuit to have repetitive patterns).

Let x and y be two functions, as specified below, which are to be realized on this tiny *Dharma* architecture.

$$x = acd + bcd + cdgh'i' + cdg'hi' + e + f$$
$$y = cde'f'gh'i'j + cde'f'g'hi'j$$

The task of the synthesizer is to transform this input specification in a manner such that the resulting network has the repetitive patterns shown in Figure 13.3. Consider

the following solution.

$$
\begin{aligned}
x &= u + v \\
y &= vw \\
p &= a + b \\
q &= cd \\
o &= gh' + g'h \\
r &= pq \\
s &= e + f \\
t &= oq \\
u &= r + s \\
v &= i't \\
w &= js'
\end{aligned}
$$

Figure 13.4(a) shows this network in graphical form. Figure 13.4(b) identifies the patterns of Figure 13.3 in this network, and this guides which function will be implemented in which DLM, and at what time instant.

13.2.3 Remarks

The example shows just a couple of repetitive patterns. Several more are possible, especially when the number of DLM inputs are more than 2. The synthesizer requirements are similar to those of a library-based mapper, which matches patterns in a given input specification. However, the pattern matching is more involved here, since patterns share common nodes. For example, the two patterns of Figure 13.3 share *D2*. Patterns that can be easily handled by the synthesizer must be chosen, and these set the location of the fixed connections.

Another possibility to investigate is to have not only repetitive connection patterns, but also repetitive function patterns. For example, if the function $a + b'$ is used in every level, then the DLM realizing this function need have only configuration bits for one level,

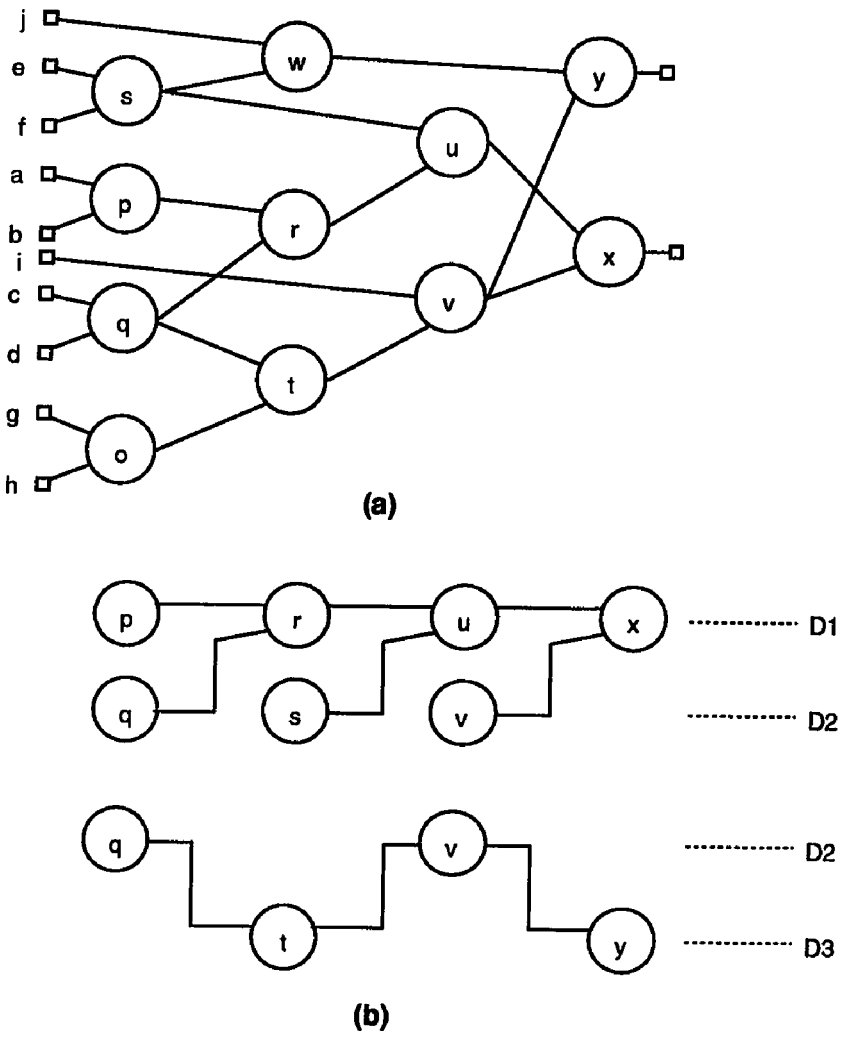


Figure 13.4: An example DAG with repetitive structures.

instead of for every level. A heterogenous DLM array (as in section 10.1.4) will be useful if a synthesizer with such a capability can be developed.

Chapter 14

Conclusions

This thesis addressed the use of field programmable devices in applications with timing performance constraints. FPLDs provide rapid *time-to-market* capabilities, and at a *low cost* – two key features for the success of ASICs. Existing FPLDs are used more for prototyping and for implementing circuits with non-critical timing specifications. With the increasing need for faster circuits, FPLDs and their accompanying software must be able to implement circuits with timing constraints on programmable devices; and this work addresses just such a need.

In Part I, we presented performance-driven synthesis techniques for existing LUT based FPGAs. Two issues were addressed independently - routing feasibility, and timing performance. We presented the **RFR** algorithm which performs synthesis (mapping), placement and routing in tandem within a simulated annealing algorithm. Experimental results showed how the approach reduced the number of unrouted nets. We showed how the performance-driven mapping and area-delay trade-off techniques of standard cell library-based mappers can be extended to LUTs. The concept of a two-input LUT primitive cell, which reduces the library size for LUT libraries by several orders of magnitude, was introduced. The special structure of the LUT library patterns was exploited to reduce effective library size even further, to $O(K^2)$, for a K-input LUT library. Experiments showed that this **dpmap** algorithm produces faster circuits, and provides useful area-delay trade-off features. **dpmap** can be extended in several directions. Libraries for the recently introduced LUT FPGAs need to be developed. A three-input LUT can also be used in addition to the 2-input LUT as a primitive cell. Integrating placement with library-based mapping can give better estimates of net delay (as compared to the linear delay model used in **dpmap**).

The initial decomposition into two-input nodes, which is the starting point for the **dpmap** algorithm, is an open problem and merits further study.

In Part II, we introduced a new FPLD architecture, *Dharma*. *Dharma* is based on the concept of time-sharing of resources in an effort to have a fully routable, CAD friendly FPLD with predictable timing performance and silicon efficiency. Real-time reconfiguration of logic and routing resources is used to implement a circuit in a folded pipe-line fashion, pipe-lining at the gate level. We explained the operation of *Dharma* and discussed several possible variations of a basic *Dharma* architecture. We introduced a synthesis scheme for *Dharma* based on temporal partitioning and presented experimental results. Area and timing analyses of *Dharma* were done, and we showed that even a worst-case *Dharma* device has advantages, both in terms of area and timing. The *Dharma* architecture presents many interesting challenges and avenues for research. Firstly, a prototype device needs to be built. *Dharma* is a memory intensive design, amenable to many technologies. The DIA has to be made small and fast, and the *Copier* idea (see Section 10.2.5) may be helpful. Sparse crossbars, along with hard-wired connections give area and speed benefits, but require intelligent synthesis tools to ensure routability. A family of *Dharma* devices, each differing in DIA and DLM architecture can be fabricated, as shown in Figure 1.3. Secondly, as outlined in Chapter 13, development of logic synthesis algorithms for *Dharma* can give better timing performance and reduce resource utilization. Thirdly, the *Dharma* idea can be extended to the system level, instead of just at the device level, to allow large sections of a system to be reconfigured while other sections are performing useful tasks.

FPLDs have so far been used mainly to implement "glue logic". Rapid time-to-market, and low cost are only two of the features of programmable devices that have been exploited so far. The third feature of field programmable devices is their ability to be reconfigured in real-time. The *Dharma* architecture exploited this feature to its advantage. But *Dharma* has only scratched the surface of potential uses for real-time reconfiguration. Properly put to use, real-time reconfiguration (*dynamic/virtual* logic, or *logic recycling*) can foster new computing paradigms, and create a revolution in logic design.

Bibliography

[Actel 91] *ACT family field programmable gate array databook*, Actel Corporation, March 1991.

[Abouzeid 90] P. Abouzeid, K. Sakouti, G. Saucier and F. Poirot, "Multilevel synthesis minimizing the routing factor," *Proc. 27th ACM/IEEE Design Automation Conference*, pp. 365-368, 1990.

[Aho 77] A. V. Aho, M. R. Garey and F. K. Hwang, "Rectilinear Steiner trees: efficient special-case algorithms," *Network*, vol. 7, pp. 37-58, 1977.

[Alfke] Peter Alfke, Xilinx, Inc. Personal Communication.

[Alford 89] R. C. Alford, *Programmable logic designer's guide*, Howard W. Sams and Co., 1989.

[Altera 85] *Altera: EPLD handbook*, Altera Corporation, 1985.

[Bhat 93] N. B. Bhat, K. Chaudhary and E. S. Kuh, *Performance-oriented fully routable dynamic architecture for a field programmable logic device*, Technical report, Electronics Research Laboratory, U. C. Berkeley, UCB/ERL Memo No. M93/42, June 1993.

[Bhat 92a] N. B. Bhat and D. D. Hill, "Routable technology mapping for LUT FPGAs," *Proc. ICCD '92*, pp. 95-98, Oct. 1992.

[Bhat 92] N. B. Bhat and D. D. Hill, "Routable technology mapping for FPGAs," *Proc. FPGA '92*, pp. 143-147, Feb. 1992.

[Brayton 90] R. K. Brayton, G. D. Hachtel and A. L. Sangiovanni-Vincentelli, "Multilevel logic synthesis," Proc. of the IEEE, vol. 78, no. 2, pp. 264-300, Feb. 1990.

[Brayton 87] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A. Wang, "MIS: a multiple-level logic optimization system," *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, no. 6, pp. 1062-1081, Nov. 1987.

[Brown 92] S. D. Brown, R. J. Francis, J. Rose and Z. G. Vranesic, *Field-programmable gate arrays*, Kluwer Academic Publishers, 1992.

[Chaudhary 93] K. Chaudhary, Personal Communication.

[Chaudhary 92] K. Chaudhary and M. Pedram, "A near optimal algorithm for technology mapping minimizing area under delay constraints," *Proc. 29th-Design Automation Conference*, pp. 492-498, 1992.

[Chen 92] K. C. Chen, "Logic minimization of lookup-table based FPGAs," *Proc. FPGA'92*, pp. 71-76, Feb. 1992.

[Cohoon 90] J. P. Cohoon, D. S. Richard and J. S. Salowe, "An optimal Steiner tree algorithm for a net whose terminal lie on the perimeter of a rectangle," *IEEE Transactions on Computer-Aided Design*, vol. 9, no. 4, pp. 398-407, April 1990.

[Cong 92] J. Cong and Y. Ding, "An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA design," *Proc. Int. Conf. CAD (ICCAD-92)*, pp. 48-53, Nov. 1992.

[Detjens 87] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli and A. Wang, "Technology mapping in MIS," *Proc. Int. Conf. CAD (ICCAD-87)*, pp. 116-119, Nov. 1987.

[Donath 88] W. E. Donath, *Logic partitioning* in *Physical design automation of VLSI systems*, B. Preas and M. Lorenzetti, editors. The Benjamin/Cummings Publisher Company, Menlo Park, California 94025, 1988.

[Ercolani 91] S. Ercolani and G. De Micheli, "Technology mapping for electrically programmable gate arrays," *Proc. 28th Design Automation Conference*, pp. 234-239, 1991.

[Filo 91] D. Filo, J. C. Yang, F. Mailhot and G. De Micheli, "Technology mapping for a two-output RAM-based field programmable gate array," *Proc. EDAC*, pp. 534-538, Feb. 1991.

[Fujita 91] M. Fujita and Y. Matsunaga, "Multi-level minimization based on minimal support and its application to the minimization of look-up table type FPGAs,"*Proc. Int. Conf. CAD (ICCAD-91)*, pp. 560-563, Nov. 1991.

[Francis 92] R. J. Francis, "A tutorial on logic synthesis for lookup- table based FPGAs," *Proc. Int. Conf. CAD (ICCAD-92)*, pp. 40-47, Nov. 1992.

[Francis 91a] R. Francis, J. Rose and Z. Vranesic, "Technology mapping of lookup table-based FPGAs for performance," *Proc. Int. Conf. CAD (ICCAD-91)*, pp. 568-571, Nov. 1991.

[Francis 91] R. Francis, J. Rose and Z. Vranesic, "Chortle-crf: Fast technology mapping for lookup table-based FPGAs," *Proc. 28th ACM/IEEE Design Automation Conference*, pp. 227-233, 1991.

[Francis 90] R. Francis, J. Rose and K. Chung, "Chortle: a technology mapping program for lookup table-based field programmable gate arrays," *Proc. 27th Design Automation Conference*, pp. 613-619, 1990.

[Garey 77] M. R. Garey and D. S. Johnson, "The rectilinear Steiner tree problem is NP-complete," *SIAM Journal of Applied Mathematics*, vol. 32, no. 4, pp. 37-58, 1977.

[Hanan 66] M. Hanan, "On Steiner's problem with rectilinear distance," *SIAM Journal of Applied Mathematics*, vol. 14, no. 2, pp. 255-265, 1966.

[Hastie 90] N. Hastie and R. Cliff, "The implementation of hardware subroutines on field programmable gate arrays," *Proc. CICC*, pp. 31.4.1-31.4.4, May 1990.

[Ho 89] J. Ho, G. Vijayan and C. K. Wong, "A new approach to the rectilinear steiner tree problem," *Proc. 26th ACM/IEEE Design Automation Conference*, pp. 161-166, 1989.

[Ho 90] J. Ho, G. Vijayan and C. K. Wong, "New algorithms for the rectilinear Steiner tree problem," *IEEE Transactions on Computer-Aided Design*, vol. 9, no. 2, pp. 185-193, Feb. 1990.

[Hodges 88] D. Hodges and H. G. Jackson, *Analysis and Design of Digital Integrated Circuits*, McGraw Hill, Second Edition, 1988.

[Horng 92] C-H Horng, Vice-President, Research and Development, I-Cube Design Systems, Inc. Personal Communication. I-Cube markets large (80 by 80) single-chip crossbar switches.

[Hsieh 90] H-C. Hsieh, W. S. Carter, J. Ja, E. Cheung, S. Schreifels, C. Erickson, P. Freidin, L. Tinkey and R. Kanazawa, "Third-generation architecture boosts speed and density of field-programmable gate arrays," *Proc. Custom Integrated Circuits Conference (CICC 90)*, pp. 31.2.1-31.2.7, 1990.

[Hwang 79] F. K. Hwang, "An O(n log n) algorithm for suboptimal rectilinear Steiner trees," *IEEE Transactions on Circuits and Systems*, vol. CAS-26, no. 1, pp. 75-77, Jan. 1979.

[Karplus 91a] K. Karplus, "Xmap: a technology mapper for table-lookup field-programmable gate arrays," *Proc. 28th Design Automation Conference*, pp. 240-243, 1991.

[Karplus 91] K. Karplus, "Amap: a technology mapper for selector-based field-programmable gate arrays," *Proc. 28th Design Automation Conference*, pp. 244-247, 1991.

[Kernighan 70] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, vol. 49, no. 2, pp. 291-308, 1970.

[Keutzer 87] K. Keutzer, "DAGON: technology binding and local optimization by DAG matching," *Proc. 24-th Design Automation Conference*, pp. 341-347, 1987.

[Kim 93] T-S. Kim, W. Burleson and M. Ciesielski, "Logic restructuring for wave-pipelined circuits," *Proc. International Workshop on Logic Synthesis*, Tahoe City, CA, pp. 4b1-4b14, May 1993.

[Kouloheris 91] J. L. Kouloheris and A. El Gamal, "FPGA performance versus cell granularity," *Proc. CICC*, pp. 6.2.1-6.2.4, May 1991.

[Kuh 90] E. S. Kuh and T. Ohtsuki, "Recent advances in VLSI layout," *Proc. of the IEEE*, vol. 78, no. 2, pp. 237-263, Feb. 1990.

[Lee 76] J. H. Lee, N. K. Base and F. K. Hwang, "Use of Steiner's problem in suboptimal routing in rectilinear metric," *IEEE Transactions on Circuit and System*, vol. CAS-23, no. 7, pp. 470-476, July 1976.

[Lee 88] K-W. Lee and C. Sechen, "A new global router for row-based layout," *Proc. Int. Conf. CAD (ICCAD-88)*, pp. 180-183, Nov. 1988.

[Masson 77] G. M. Masson, "Binomial switching networks for concentration and distribution," *IEEE Transactions on Communication*, COM-25, no.9, pp. 873-883, Sept. 1977.

[MCNC] S. Yang, "Logic synthesis and optimization benchmarks user guide - version 3.0," Microelectronic Center of North Carolina, Jan. 1991.

[Murgai 91a] R. Murgai, N. Shenoy, R. K. Brayton and A. Sangiovanni-Vincentelli, "Performance directed synthesis for table look up programmable gate arrays," *Proc. Int. Conf. CAD (ICCAD-91)*, pp. 572-575, Nov. 1991.

[Murgai 91] R. Murgai, N. Shenoy, R. K. Brayton and A. Sangiovanni-Vincentelli,"Improved logic synthesis algorithms for table look up architectures," *Proc. Int. Conf. CAD (ICCAD-91)*, pp. 564-567, Nov. 1991.

[Murgai 90] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton and A. Sangiovanni-Vincentelli, "Logic synthesis for programmable gate arrays," *Proc. 27th Design Automation Conference*, pp. 620-625, 1990.

[ORCA 93] *Optimized Reconfigurable Cell Array (ORCA) Series Field-Programmable Gate Arrays*, AT&T Microelectronics, Advance Data Sheet, Feb 1993.

[Park 91] I-C. Park, C-M. Kyung, "Fast and near optimal scheduling in automatic data path synthesis," *Proc. 28th Design Automation Conference*, pp. 680-685, 1991.

[Pedram 91a] M. Pedram and N. Bhat, "Layout driven logic decomposition/restructuring," *Proc. Int. Conf. CAD (ICCAD-91)*, pp. 134-137, Nov. 1991.

[Pedram 91] M. Pedram and N. Bhat, "Layout driven technology mapping," *Proc. 28th Design Automation Conference*, pp. 99-105, 1991.

[Quickturn 93] *A new approach to design verification and system integration: computer aided prototyping*, Quickturn Systems Inc., 1993.

[Saldanha 89] A. Saldanha, A. Wang, R. Brayton and A. Sangiovanni-Vincentelli, "Multi-level logic simplification using don't cares and filters," *Proc. 26th Design Automation Conference*, 1989.

[Sawkar 92] P. Sawkar, D. Thomas, "Area and delay mapping for table look-up based field programmable gate arrays," *Proc. 29th Design Automation Conference*, pp. 368-373, 1992.

[Schlag 92] M. Schlag, J. Kong and P. K. Chan, "Routability-driven technology mapping for lookup-table-based FPGAs," *Proc. ICCD*, pp. 86-90, Oct. 1992.

[Schlichtmann 92] U. Schlichtmann, F. Brglez and M. Hermann, "Characterization of Boolean functions for rapid matching in EPGA technology mapping," *Proc. 29th Design Automation Conference*, pp. 374-379, 1992.

[Sechen 88] C. Sechen, "Chip-planning, placement, and global routing of macro/custom cell integrated circuits using simulated annealing," *Proc. 25th Design Automation Conference*, pp. 73-80, 1988.

[Sentovich 92] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton and A. L. Sangiovanni-Vincentelli, "Sequential circuit design using synthesis and optimization," *Proc. ICCD*, pp. 328-33, Oct. 1992.

[Singh 92] S. Singh, J. Rose, P. Chow and D. Lewis, "The effect of logic block architecture on FPGA performance," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 3, pp. 281-287, Mar. 1992.

[Small 91] C. S. Small, "High-density PLD architectures: family tree sorts out high-density PLDs," *EDN*, pp. 75-80, Sept. 1991.

[Touati 90] H. J. Touati, *Performance-oriented technology mapping*, Ph.D. Thesis, U. C. Berkeley, UCB/ERL Memo No. M90/109, Nov. 1990.

[Trimberger 92] S. Trimberger and M. Chene, "Placement-based partitioning for lookup-table-based FPGAs," *Proc. ICCD '92*, pp. 91-94, Oct. 1992.

[Trimberger 92a] S. Trimberger, "A small complete mapping library for lookup-table-based FPGAs," *Proc. 2nd International Workshop on Field-Programmable Logic and Applications*, Aug. 1992.

[Tseng 92] B. Tseng, J. Rose and S. Brown, "Improving FPGA routing architectures using architecture and CAD interactions," *Proc. ICCD '92*, pp. 99-104, Oct. 1992.

[Vecchi 83] M. P. Vecchi and S. Kirkpatrick, "Global wiring by simulated annealing," *IEEE Transactions on Computer-Aided Design*, vol. CAD-2, no. 4, pp. 215-222, Oct. 1983.

[Waugh 91] T. C. Waugh and A. Kopser, "Field programmable gate array key to reconfigurable array outperforming supercomputers," *Proc. CICC*, pp. 6.6.1-6.6.4, May 1991.

[Woo 91] N. Woo, "A heuristic method for FPGA technology mapping based on edge visibility," *Proc. 28th ACM/IEEE Design Automation Conference*, pp. 248-251, 1991.

[Xilinx 89] *Xilinx: The programmable gate array data book*, Xilinx Inc., 1989.

[Yang 72] Y. Y. Yang and O. Wing, "Suboptimal algorithm for a wire routing problem," *IEEE Transactions on Circuits Theory*, pp. 508-511, Sept. 1972.

[Yeh91] C. W. Yeh and C. K .Cheng, "A general purpose multiple way partitioning algorithm," *Proc. 28th ACM/IEEE Design Automation Conference*, pp. 421-426, 1991.