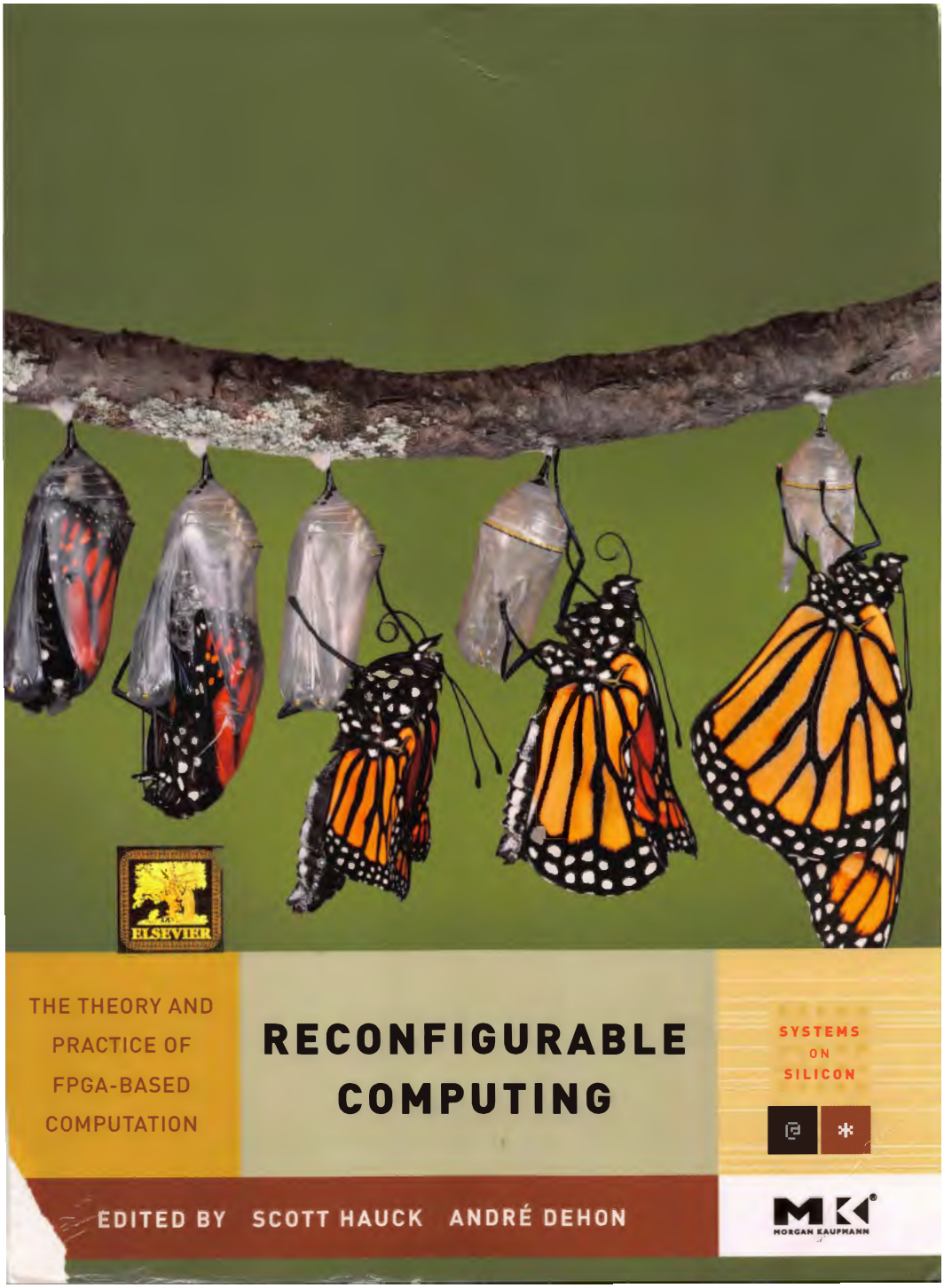
Homayoun

Reference 25



RECONFIGURABLE COMPUTING

The Morgan Kaufmann Series in Systems on Silicon

Series Editor: Wayne Wolf, Georgia Institute of Technology

The Designer's Guide to VHDL, Second Edition
Peter J. Ashenden

The System Designer's Guide to VHDL-AMS
Peter J. Ashenden, Gregory D. Peterson, and Darrell A. Teegarden

Modeling Embedded Systems and SoCs
Axel Jantsch

ASIC and FPGA Verification: A Guide to Component Modeling
Richard Munden

Multiprocessor Systems-on-Chips
Edited by Ahmed Amine Jerraya and Wayne Wolf

Functional Verification
Bruce Wile, John Goss, and Wolfgang Roesner

Customizable and Configurable Embedded Processors
Edited by Paolo Ienne and Rainer Leupers

Networks-on-Chips: Technology and Tools
Edited by Giovanni De Micheli and Luca Benini

VLSI Test Principles & Architectures
Edited by Laung-Terng Wang, Cheng-Wen Wu, and Xiaoqing Wen

Designing SoCs with Configured Processors
Steve Leibson

ESL Design and Verification
Grant Martin, Andrew Piziali, and Brian Bailey

*Aspect-Oriented Programming with **e***
David Robinson

Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation
Edited by Scott Hauck and André DeHon

Coming Soon ...

System-on-Chip Test Architectures
Edited by Laung-Terng Wang, Charles Stroud, and Nur Touba

Verification Techniques for System-Level Design
Masahiro Fujita, Indradeep Ghosh, and Mukul Prasad

RECONFIGURABLE COMPUTING

THE THEORY AND PRACTICE OF FPGA-BASED COMPUTATION

Edited by

Scott Hauck and André DeHon

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW DELHI • NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
Morgan Kaufmann Publishers is an imprint of Elsevier

MORGAN KAUFMANN

Reconfigurable Computing
Hauck and DeHon

MORGAN KAUFMANN PUBLISHERS
An imprint of Elsevier
30 Corporate Drive, Suite 400, Burlington, MA 01803-4255

Original ISBN: 978-0-12-370522-8

First Printed in India 2011

Indian Reprint ISBN: 978-93-80931-86-9

This edition has been authorized by Elsevier for sale in the following countries: India, Pakistan, Nepal, Sri Lanka and Bangladesh. Sale and purchase of this book outside these countries is not authorized and is illegal.

Published by Elsevier, a division of Reed Elsevier India Private Limited.

Registered Office: 622, Indraprakash Building, 21 Barakhamba Road, New Delhi-110 001.
Corporate Office: 14th floor, Building No. 10B, DLF Cyber City Phase-II, Gurgaon-122 002, Haryana, India.

Printed and bound in India by Sanat Printers, Kundli-131 028

Thomas W. Fry, Samsung, Global Strategy Group, Seoul, South Korea (Chapter 27)

Maya B. Gokhale, Lawrence Livermore National Laboratory, Livermore, California (Chapter 10)

Steven A. Guccione, Cmpware, Inc., Austin, Texas (Chapters 3 and 19)

Scott Hauck, Department of Electrical Engineering, University of Washington, Seattle, Washington (Chapters 20 and 27)

K. Scott Hemmert, Computation, Computers, Information and Mathematics Center, Sandia National Laboratories, Albuquerque, New Mexico (Chapter 31)

Randy Huang, Tabula, Inc., Santa Clara, California (Chapter 9)

Brad L. Hutchings, Department of Electrical and Computer Engineering, Brigham Young University, Provo, Utah (Chapters 12 and 21)

Nachiket Kapre, Department of Computer Science, California Institute of Technology, Pasadena, California (Chapter 6)

Andreas Koch, Department of Computer Science, Embedded Systems and Applications Group, Technische Universität of Darmstadt, Darmstadt, Germany (Chapter 15)

Miriam Leeser, Department of Electrical and Computer Engineering, Northeastern University, Boston, Massachusetts (Chapter 32)

John W. Lockwood, Department of Computer Science and Engineering, Washington University in St. Louis, St. Louis, Missouri; and Department of Electrical Engineering, Stanford University, Stanford, California (Chapter 34)

Wayne Luk, Department of Computing, Imperial College, London, United Kingdom (Chapter 22)

Sharad Malik, Department of Electrical Engineering, Princeton University, Princeton, New Jersey (Chapter 29)

Yury Markovskiy, Department of Electrical Engineering and Computer Sciences, University of California–Berkeley, Berkeley, California (Chapter 9)

Margaret Martonosi, Department of Electrical Engineering, Princeton University, Princeton, New Jersey (Chapter 29)

Larry McMurchie, Synplicity Corporation, Sunnyvale, California (Chapter 17)

Brent E. Nelson, Department of Electrical and Computer Engineering, Brigham Young University, Provo, Utah (Chapters 12 and 21)

Peichen Pan, Magma Design Automation, Inc., San Jose, California (Chapter 13)

Oliver Pell, Department of Computing, Imperial College, London, United Kingdom (Chapter 22)

Stylianos Perissakis, Department of Electrical Engineering and Computer Sciences, University of California–Berkeley, Berkeley, California (Chapter 9)

Laura Pozzi, Faculty of Informatics, University of Lugano, Lugano, Switzerland (Chapter 9)

Brian C. Richards, Department of Electrical Engineering and Computer Sciences, University of California–Berkeley, Berkeley, California (Chapter 8)

Eduardo Sanchez, School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne; and Reconfigurable and Embedded Digital Systems Institute, Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud, Lausanne, Switzerland (Chapter 33)

Lesley Shannon, School of Engineering Science, Simon Fraser University, Burnaby, BC, Canada (Chapter 2)

Satnam Singh, Programming Principles and Tools Group, Microsoft Research, Cambridge, United Kingdom (Chapter 16)

Greg Stitt, Department of Computer Science and Engineering, University of California–Riverside, Riverside, California (Chapter 26)

Russell Tessier, Department of Computer and Electrical Engineering, University of Massachusetts, Amherst, Massachusetts (Chapter 30)

Keith D. Underwood, Computation, Computers, Information and Mathematics Center, Sandia National Laboratories, Albuquerque, New Mexico (Chapter 31)

Andres Upegui, Logic Systems Laboratory, School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland (Chapter 33)

Frank Vahid, Department of Computer Science and Engineering, University of California–Riverside, Riverside, California (Chapter 26)

John Wawrzynek, Department of Electrical Engineering and Computer Sciences, University of California–Berkeley, Berkeley, California (Chapters 8 and 9)

Nicholas Weaver, International Computer Science Institute, Berkeley, California (Chapter 18)

Joseph Yeh, Lincoln Laboratory, Massachusetts Institute of Technology, Lexington, Massachusetts (Chapter 9)

Peixin Zhong, Department of Electrical and Computer Engineering, Michigan State University, East Lansing, Michigan (Chapter 29)

PREFACE

In the two decades since field-programmable gate arrays (FPGAs) were introduced, they have radically changed the way digital logic is designed and deployed. By marrying the high performance of application-specific integrated circuits (ASICs) and the flexibility of microprocessors, FPGAs have made possible entirely new types of applications. This has helped FPGAs supplant both ASICs and digital signal processors (DSPs) in some traditional roles.

To make the most of this unique combination of performance and flexibility, designers need to be aware of both hardware and software issues. Thus, an FPGA user must think not only about the gates needed to perform a computation but also about the software flow that supports the design process. The goal of this book is to help designers become comfortable with these issues, and thus be able to exploit the vast opportunities possible with reconfigurable logic.

We have written *Reconfigurable Computing* as a tutorial and as a reference on the wide range of concepts that designers must understand to make the best use of FPGAs and related reconfigurable chips—including FPGA architectures, FPGA logic applications, and FPGA CAD tools—and the skills they must have for optimizing a computation. It is targeted particularly toward those who view FPGAs not just as cheap, slow ASIC gates or as a means of prototyping before the "real" hardware is created, but are interested in evaluating or embracing the substantial advantages reprogrammable devices offer over other technologies. However, readers who focus primarily on ASIC- or CPU-based implementations will learn how FPGAs can be a useful addition to their normal skill set. For some traditional designers this book may even serve as an entry point into a completely new way of handling their design problems.

Because we focus on both hardware and software systems, we expect readers to have a certain level of familiarity with each technology. On the hardware side, we assume that readers have a basic knowledge of digital logic design, including understanding concepts such as gates (including multiplexers, flip-flops, and RAM), binary number systems, and simple logic optimization. Knowledge of hardware description languages, such as Verilog or VHDL, is also helpful. We also assume that readers have basic knowledge of computer programming, including simple data structures and algorithms. In sum, this book is appropriate for most readers with a background in electrical engineering, computer science, or computer engineering. It can also be used as a text in an upper-level undergraduate or introductory graduate course within any of these disciplines.

No one book can hope to cover every possible aspect of FPGAs exhaustively. Entire books could be (and have been) written about each of the concepts that are discussed in the individual chapters here. Our goal is to provide a good working knowledge of these concepts, as well as abundant references for those who wish to dig deeper.

Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation is divided into six major parts—hardware, programming, compilation/mapping, application development, case studies, and future trends. Once the introduction has been read, the parts can be covered in any order. Alternatively, readers can pick and choose which parts they wish to cover. For example, a reader who wants to focus on CAD for FPGAs might skip hardware and application development, while a reader who is interested mostly in the use of FPGAs might focus primarily on application development.

Part V is made up of self-contained overviews of specific, important applications, which can be covered in any order or can be sprinkled throughout a course syllabus. The part introduction lists the chapters and concepts relevant to each case study and so can be used as a guide for the reader or instructor in selecting relevant examples.

One final consideration is an explanation of how this book was written. Some books are created by a single author or a set of coauthors who must stretch to cover all aspects of a given topic. Alternatively, an edited text can bring together contributors from each of the topic areas, typically by bundling together standalone research papers. Our book is a bit of a hybrid. It was constructed from an overall outline developed by the primary authors, Scott Hauck and André DeHon. The chapters on the chosen topics were then written by noted experts in these areas, and were carefully edited to ensure their integration into a cohesive whole. Our hope is that this brings the benefits of both styles of traditional texts, with the reader learning from the main experts on each topic, yet still delivering a well-integrated text.

Acknowledgments

While Scott and André handled the technical editing, this book also benefited from the careful help from the team at Elsevier/Morgan Kaufmann. Wayne Wolf first proposed the concept of this book to us. Chuck Glaser, ably assisted by Michele Cronin and Matthew Cater, was instrumental in resurrecting the project after it had languished in the concept stage for several years and in pushing it through to completion. Just as important were the efforts of the production group at Elsevier/Morgan Kaufmann who did an excellent job of copyediting, proofreading, integrating text and graphics, laying out, and all the hundreds of little details crucial to bringing a book together into a polished whole. This was especially true for a book like this, with such a large list of contributors. Specifically, Marilyn E. Rash helped drive the whole production process and was supported by Dianne Wood, Jodie Allen, and Steve Rath. Without their help there is no way this monumental task ever would have been finished. A big thank you to all.

Scott Hauck
André DeHon

INTRODUCTION

In the computer and electronics world, we are used to two different ways of performing computation: hardware and software. Computer hardware, such as application-specific integrated circuits (ASICs), provides highly optimized resources for quickly performing critical tasks, but it is permanently configured to only one application via a multimillion-dollar design and fabrication effort. Computer software provides the flexibility to change applications and perform a huge number of different tasks, but is orders of magnitude worse than ASIC implementations in terms of performance, silicon area efficiency, and power usage.

Field-programmable gate arrays (FPGAs) are truly revolutionary devices that blend the benefits of both hardware and software. They implement circuits just like hardware, providing huge power, area, and performance benefits over software, yet can be reprogrammed cheaply and easily to implement a wide range of tasks. Just like computer hardware, FPGAs implement computations spatially, simultaneously computing millions of operations in resources distributed across a silicon chip. Such systems can be hundreds of times faster than microprocessor-based designs. However, unlike in ASICs, these computations are programmed into the chip, not permanently frozen by the manufacturing process. This means that an FPGA-based system can be programmed and reprogrammed many times.

Sometimes reprogramming is merely a bug fix to correct faulty behavior, or it is used to add a new feature. Other times, it may be carried out to reconfigure a generic computation engine for a new task, or even to reconfigure a device during operation to allow a single piece of silicon to simultaneously do the work of numerous special-purpose chips.

However, merging the benefits of both hardware and software does come at a price. FPGAs provide nearly all of the benefits of software flexibility and development models, and nearly all of the benefits of hardware efficiency—but not quite. Compared to a microprocessor, these devices are typically several orders of magnitude faster and more power efficient, but creating efficient programs for them is more complex. Typically, FPGAs are useful only for operations that process large streams of data, such as signal processing, networking, and the like. Compared to ASICs, they may be 5 to 25 times worse in terms of area, delay, and performance. However, while an ASIC design may take months to years to develop and have a multimillion-dollar price tag, an FPGA design might only take days to create and cost tens to hundreds of dollars. For systems that do not require the absolute highest achievable performance or power efficiency, an FPGA's development simplicity and the ability to easily fix bugs and upgrade functionality make them a compelling design alternative. For many tasks, and particularly for beginning electronics designers, FPGAs are the ideal choice.

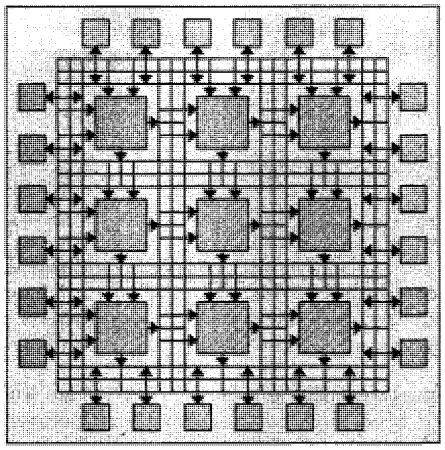


FIGURE I.1 ■ An abstract view of an FPGA; logic cells are embedded in a general routing structure.

Figure I.1 illustrates the internal workings of a field-programmable gate array, which is made up of logic blocks embedded in a general routing structure. This *array* of logic *gates* is the *G* and *A* in *FPGA*. The logic blocks contain processing elements for performing simple combinational logic, as well as flip-flops for implementing sequential logic. Because the logic units are often just simple memories, any Boolean combinational function of perhaps five or six inputs can be implemented in each logic block. The general routing structure allows arbitrary wiring, so the logical elements can be connected in the desired manner.

Because of this generality and flexibility, an FPGA can implement very complex circuits. Current devices can compute functions on the order of millions of basic gates, running at speeds in the hundreds of Megahertz. To boost speed and capacity, additional, special elements can be embedded into the array, such as large memories, multipliers, fast-carry logic for arithmetic and logic functions, and even complete microprocessors. With these predefined, fixed-logic units, which are fabricated into the silicon, FPGAs are capable of implementing complete systems in a single programmable device.

The logic and routing elements in an FPGA are controlled by programming points, which may be based on antifuse, Flash, or SRAM technology. For reconfigurable computing, SRAM-based FPGAs are the preferred option, and in fact are the primary style of FPGA devices in the electronics industry as a whole. In these devices, every routing choice and every logic function is controlled by a simple memory bit. With all of its memory bits programmed, by way of a configuration file or bitstream, an FPGA can be configured to implement the user's desired function. Thus, the configuration can be carried out quickly and

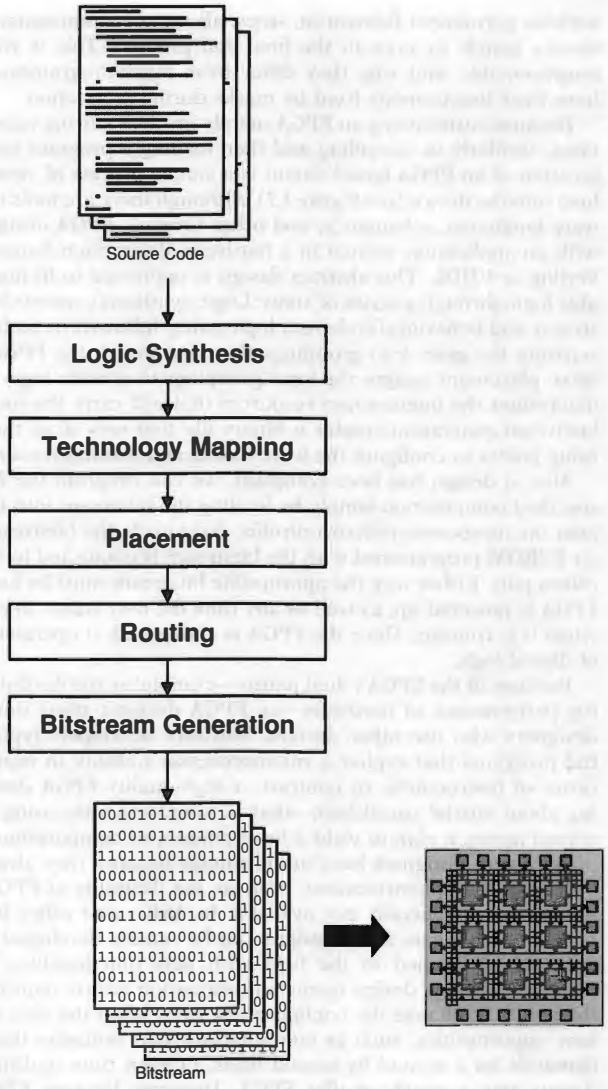


FIGURE I.2 ■ A typical FPGA mapping flow.

without permanent fabrication steps, allowing customization at the user's electronics bench, or even in the final end product. This is why FPGAs are *field programmable,* and why they differ from mask-programmable devices, which have their functionality fixed by masks during fabrication.

Because customizing an FPGA merely involves storing values to memory locations, similarly to compiling and then loading a program onto a computer, the creation of an FPGA-based circuit is a simple process of creating a bitstream to load into the device (see Figure I.2). Although there are tools to do this from software languages, schematics, and other formats, FPGA designers typically start with an application written in a hardware description language (HDL) such as Verilog or VHDL. This abstract design is optimized to fit into the FPGA's available logic through a series of steps: Logic synthesis converts high-level logic constructs and behavioral code into logic gates, followed by technology mapping to separate the gates into groupings that best match the FPGA's logic resources. Next, placement assigns the logic groupings to specific logic blocks and routing determines the interconnect resources that will carry the user's signals. Finally, bitstream generation creates a binary file that sets all of the FPGA's programming points to configure the logic blocks and routing resources appropriately.

After a design has been compiled, we can program the FPGA to perform a specified computation simply by loading the bitstream into it. Typically either a host microprocessor/microcontroller downloads the bitstream to the device, or an EPROM programmed with the bitstream is connected to the FPGA's configuration port. Either way, the appropriate bitstream must be loaded every time the FPGA is powered up, as well as any time the user wants to change the circuitry when it is running. Once the FPGA is configured, it operates as a custom piece of digital logic.

Because of the FPGA's dual nature—combining the flexibility of software with the performance of hardware—an FPGA designer must think differently from designers who use other devices. Software developers typically write sequential programs that exploit a microprocessor's ability to rapidly step through a series of instructions. In contrast, a high-quality FPGA design requires thinking about spatial parallelism—that is, simultaneously using multiple resources spread across a chip to yield a huge amount of computation.

Hardware designers have an advantage because they already think in terms of hardware implementations; even so, the flexibility of FPGAs gives them new opportunities generally not available in ASICs and other fixed devices. Field-programmable gate array designs can be rapidly developed and deployed, and even reprogrammed in the field with new functionality. Thus, they do not demand the huge design teams and validation efforts required for ASICs. Also, the ability to change the configuration, even when the device is running, yields new opportunities, such as computations that optimize themselves to specific demands on a second-by-second basis, or even time multiplexing a very large design onto a much smaller FPGA. However, because FPGAs are noticeably slower and have lower capacity than ASICs, designers must carefully optimize their design to the target device.

FPGAs are a very flexible medium, with unique opportunities and challenges. The goal of *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation* is to introduce all facets of FPGA-based systems—both positive and problematic. It is organized into six major parts:

- Part I introduces the hardware devices, covering both generic FPGAs and those specifically optimized for reconfigurable computing (Chapters 1 through 4).
- Part II focuses on programming reconfigurable computing systems, considering both their programming languages and programming models (Chapters 5 through 12).
- Part III focuses on the software mapping flow for FPGAs, including each of the basic CAD steps of Figure I.2 (Chapters 13 through 20).
- Part IV is devoted to application design, covering ways to make the most efficient use of FPGA logic (Chapters 21 through 26). This part can be viewed as a finishing school for FPGA designers because it highlights ways in which application development on an FPGA is different from both software programming and ASIC design.
- Part V is a set of case studies that show complete applications of reconfigurable logic (Chapters 27 through 35).
- Part VI contains more advanced topics, such as theoretical models and metric for reconfigurable computing, as well as defect and fault tolerance and the possible synergies between reconfigurable computing and nanotechnology (Chapters 36 through 38).

As the 38 chapters that follow will show, the challenges that FPGAs present are significant. However, the effort entailed in surmounting them is far outweighed by the unique opportunities these devices offer to the field of computing technology.

CONTENTS

List of Contributors xx
Preface xxiii
Introduction xxv

Part I: Reconfigurable Computing Hardware 1

1 Device Architecture 3

1.1 Logic—The Computational Fabric 3
1.1.1 Logic Elements 4
1.1.2 Programmability 6
1.2 The Array and Interconnect 6
1.2.1 Interconnect Structures 7
1.2.2 Programmability 12
1.2.3 Summary 12
1.3 Extending Logic 12
1.3.1 Extended Logic Elements 12
1.3.2 Summary 16
1.4 Configuration 16
1.4.1 SRAM 16
1.4.2 Flash Memory 17
1.4.3 Antifuse 17
1.4.4 Summary 18
1.5 Case Studies 18
1.5.1 Altera Stratix 19
1.5.2 Xilinx Virtex-II Pro 23
1.6 Summary 26
References 27

2 Reconfigurable Computing Architectures 29

2.1 Reconfigurable Processing Fabric Architectures 30
2.1.1 Fine-grained 30
2.1.2 Coarse-grained 32
2.2 RPF Integration into Traditional Computing Systems 35
2.2.1 Independent Reconfigurable Coprocessor Architectures . . 36
2.2.2 Processor + RPF Architectures 40
2.3 Summary and Future Work 44
References 45

3 Reconfigurable Computing Systems 47

3.1 Early Systems 47
3.2 PAM, VCC, and Splash 49
3.2.1 PAM 49
3.2.2 Virtual Computer 50
3.2.3 Splash 51

3.3 Small-scale Reconfigurable Systems 52
3.3.1 PRISM 53
3.3.2 CAL and XC6200 53
3.3.3 Cloning 54
3.4 Circuit Emulation 54
3.4.1 AMD/Intel 55
3.4.2 Virtual Wires 56
3.5 Accelerating Technology 56
3.5.1 Teramac 57
3.6 Reconfigurable Supercomputing 59
3.6.1 Cray, SRC, and Silicon Graphics 60
3.6.2 The CMX-2X 60
3.7 Non-FPGA Research 61
3.8 Other System Issues 61
3.9 The Future of Reconfigurable Systems 62
References 63

4 Reconfiguration Management 65

4.1 Reconfiguration 66
4.2 Configuration Architectures 66
4.2.1 Single-context 67
4.2.2 Multi-context 68
4.2.3 Partially Reconfigurable 70
4.2.4 Relocation and Defragmentation 71
4.2.5 Pipeline Reconfigurable 73
4.2.6 Block Reconfigurable 74
4.2.7 Summary 75
4.3 Managing the Reconfiguration Process 76
4.3.1 Configuration Grouping 76
4.3.2 Configuration Caching 77
4.3.3 Configuration Scheduling 77
4.3.4 Software-based Relocation and Defragmentation 79
4.3.5 Context Switching 80
4.4 Reducing Configuration Transfer Time 80
4.4.1 Architectural Approaches 81
4.4.2 Configuration Compression 81
4.4.3 Configuration Data Reuse 82
4.5 Configuration Security 82
4.6 Summary 83
References 84

Part II: Programming Reconfigurable Systems 87

5 Compute Models and System Architectures 91

5.1 Compute Models 93
5.1.1 Challenges 93
5.1.2 Common Primitives 97
5.1.3 Dataflow 98
5.1.4 Sequential Control 103

5.1.5 Data Parallel . . . 105
5.1.6 Data-centric . . . 105
5.1.7 Multi-threaded . . . 106
5.1.8 Other Compute Models . . . 106
5.2 System Architectures . . . 107
5.2.1 Streaming Dataflow . . . 107
5.2.2 Sequential Control . . . 110
5.2.3 Bulk Synchronous Parallelism . . . 118
5.2.4 Data Parallel . . . 119
5.2.5 Cellular Automata . . . 122
5.2.6 Multi-threaded . . . 123
5.2.7 Hierarchical Composition . . . 125
References . . . 125

6 Programming FPGA Applications in VHDL 129

6.1 VHDL Programming . . . 130
6.1.1 Structural Description . . . 130
6.1.2 RTL Description . . . 133
6.1.3 Parametric Hardware Generation . . . 136
6.1.4 Finite-state Machine Datapath Example . . . 138
6.1.5 Advanced Topics . . . 150
6.2 Hardware Compilation Flow . . . 150
6.2.1 Constraints . . . 152
6.3 Limitations of VHDL . . . 153
References . . . 153

7 Compiling C for Spatial Computing 155

7.1 Overview of How C Code Runs on Spatial Hardware . . . 156
7.1.1 Data Connections between Operations . . . 157
7.1.2 Memory . . . 157
7.1.3 If-then-else Using Multiplexers . . . 158
7.1.4 Actual Control Flow . . . 159
7.1.5 Optimizing the Common Path . . . 161
7.1.6 Summary and Challenges . . . 162
7.2 Automatic Compilation . . . 162
7.2.1 Hyperblocks . . . 164
7.2.2 Building a Dataflow Graph for a Hyperblock . . . 164
7.2.3 DFG Optimization . . . 169
7.2.4 From DFG to Reconfigurable Fabric . . . 173
7.3 Uses and Variations of C Compilation to Hardware . . . 175
7.3.1 Automatic HW/SW Partitioning . . . 175
7.3.2 Programmer Assistance . . . 176
7.4 Summary . . . 180
References . . . 180

8 Programming Streaming FPGA Applications Using Block Diagrams in Simulink 183

8.1 Designing High-performance Datapaths Using Stream-based Operators 184
8.2 An Image-processing Design Driver 185
8.2.1 Converting RGB Video to Grayscale 185
8.2.2 Two-dimensional Video Filtering 187
8.2.3 Mapping the Video Filter to the BEE2 FPGA Platform . . 191
8.3 Specifying Control in Simulink 194
8.3.1 Explicit Controller Design with Simulink Blocks 194
8.3.2 Controller Design Using the Matlab M Language 195
8.3.3 Controller Design Using VHDL or Verilog 197
8.3.4 Controller Design Using Embedded Microprocessors . . . 197
8.4 Component Reuse: Libraries of Simple and Complex Subsystems . 198
8.4.1 Signal-processing Primitives 198
8.4.2 Tiled Subsystems 198
8.5 Summary 201
References 202

9 Stream Computations Organized for Reconfigurable Execution 203

9.1 Programming 205
9.1.1 Task Description Format 205
9.1.2 C++ Integration and Composition 206
9.2 System Architecture and Execution Patterns 208
9.2.1 Stream Support 209
9.2.2 Phased Reconfiguration 210
9.2.3 Sequential versus Parallel 211
9.2.4 Fixed-size and Standard I/O Page 211
9.3 Compilation 212
9.4 Runtime 213
9.4.1 Scheduling 213
9.4.2 Placement 215
9.4.3 Routing 215
9.5 Highlights 217
References 217

10 Programming Data Parallel FPGA Applications Using the SIMD/Vector Model 219

10.1 SIMD Computing on FPGAs: An Example 219
10.2 SIMD Processing Architectures 221
10.3 Data Parallel Languages 222
10.4 Reconfigurable Computers for SIMD/Vector Processing 223
10.5 Variations of SIMD/Vector Computing 226
10.5.1 Multiple SIMD Engines 226
10.5.2 A Multi-SIMD Coarse-grained Array 228
10.5.3 SPMD Model 228

10.6 Pipelined SIMD/Vector Processing 228
10.7 Summary 229
References 230

11 Operating System Support for Reconfigurable Computing 231

11.1 History 232
11.2 Abstracted Hardware Resources 234
11.2.1 Programming Model 234
11.3 Flexible Binding 236
11.3.1 Install Time Binding 236
11.3.2 Runtime Binding 237
11.3.3 Fast CAD for Flexible Binding 238
11.4 Scheduling 239
11.4.1 On-demand Scheduling 239
11.4.2 Static Scheduling 239
11.4.3 Dynamic Scheduling 240
11.4.4 Quasi-static Scheduling 241
11.4.5 Real-time Scheduling 241
11.4.6 Preemption 242
11.5 Communication 243
11.5.1 Communication Styles 243
11.5.2 Virtual Memory 246
11.5.3 I/O 247
11.5.4 Uncertain Communication Latency 247
11.6 Synchronization 248
11.6.1 Explicit Synchronization 248
11.6.2 Implicit Synchronization 248
11.6.3 Deadlock Prevention 249
11.7 Protection 249
11.7.1 Hardware Protection 250
11.7.2 Intertask Communication 251
11.7.3 Task Configuration Protection 251
11.8 Summary 252
References 252

12 The JHDL Design and Debug System 255

12.1 JHDL Background and Motivation 255
12.2 The JHDL Design Language 257
12.2.1 Level-1 Design: Primitive Instantiation 257
12.2.2 Level-2 Design: Using the `Logic` Class and Its Provided Methods 259
12.2.3 Level-3 Design: Programmatic Circuit Generation (Module Generators) 261
12.2.4 JHDL Is a Structural Design Language 263
12.2.5 JHDL Is a Programmatic Circuit Design Language 264
12.3 The JHDL CAD System 265
12.3.1 Testbenches in JHDL 265
12.3.2 The `cvt` Class 266

12.4 JHDL's Hardware Mode 268
12.5 Advanced JHDL Capabilities 269
12.5.1 Dynamic Testbenches 269
12.5.2 Behavioral Synthesis 270
12.5.3 Advanced Debugging Capabilities 270
12.6 Summary 272
References 273

Part III: Mapping Designs to Reconfigurable Platforms 275

13 Technology Mapping 277

13.1 Structural Mapping Algorithms 278
13.1.1 Cut Generation 279
13.1.2 Area-oriented Mapping 280
13.1.3 Performance-driven Mapping 282
13.1.4 Power-aware Mapping 283
13.2 Integrated Mapping Algorithms 284
13.2.1 Simultaneous Logic Synthesis, Mapping 284
13.2.2 Integrated Retiming, Mapping 286
13.2.3 Placement-driven Mapping 287
13.3 Mapping Algorithms for Heterogeneous Resources 289
13.3.1 Mapping to LUTs of Different Input Sizes 289
13.3.2 Mapping to Complex Logic Blocks 290
13.3.3 Mapping Logic to Embedded Memory Blocks 291
13.3.4 Mapping to Macrocells 292
13.4 Summary 293
References 293

FPGA Placement 297

14 Placement for General-purpose FPGAs 299

14.1 The FPGA Placement Problem 299
14.1.1 Device Legality Constraints 300
14.1.2 Optimization Goals 301
14.1.3 Designer Placement Directives 302
14.2 Clustering 304
14.3 Simulated Annealing for Placement 306
14.3.1 VPR and Related Annealing Algorithms 307
14.3.2 Simultaneous Placement and Routing with Annealing 311
14.4 Partition-based Placement 312
14.5 Analytic Placement 315
14.6 Further Reading and Open Challenges 316
References 316

15 Datapath Composition 319

15.1 Fundamentals . . . 319
15.1.1 Regularity . . . 320
15.1.2 Datapath Layout . . . 322
15.2 Tool Flow Overview . . . 323
15.3 The Impact of Device Architecture . . . 324
15.3.1 Architecture Irregularities . . . 325
15.4 The Interface to Module Generators . . . 326
15.4.1 The Flow Interface . . . 327
15.4.2 The Data Model . . . 327
15.4.3 The Library Specification . . . 328
15.4.4 The Intra-module Layout . . . 328
15.5 The Mapping . . . 329
15.5.1 1:1 Mapping . . . 329
15.5.2 *N*:1 Mapping . . . 330
15.5.3 The Combined Approach . . . 332
15.6 Placement . . . 333
15.6.1 Linear Placement . . . 333
15.6.2 Constrained Two-dimensional Placement . . . 335
15.6.3 Two-dimensional Placement . . . 336
15.7 Compaction . . . 337
15.7.1 Selecting HWOPs for Compaction . . . 338
15.7.2 Regularity Analysis . . . 338
15.7.3 Optimization Techniques . . . 338
15.7.4 Building the Super-HWOP . . . 342
15.7.5 Discussion . . . 343
15.8 Summary and Future Work . . . 344
References . . . 344

16 Specifying Circuit Layout on FPGAs 347

16.1 The Problem . . . 347
16.2 Explicit Cartesian Layout Specification . . . 351
16.3 Algebraic Layout Specification . . . 352
16.3.1 Case Study: Batcher's Bitonic Sorter . . . 357
16.4 Layout Verification for Parameterized Designs . . . 360
16.5 Summary . . . 362
References . . . 363

17 PathFinder: A Negotiation-based, Performance-driven Router for FPGAs 365

17.1 The History of PathFinder . . . 366
17.2 The PathFinder Algorithm . . . 367
17.2.1 The Circuit Graph Model . . . 367
17.2.2 A Negotiated Congestion Router . . . 367
17.2.3 The Negotiated Congestion/Delay Router . . . 372
17.2.4 Applying A* to PathFinder . . . 373
17.3 Enhancements and Extensions to PathFinder . . . 374
17.3.1 Incremental Rerouting . . . 374

17.3.2 The Cost Function . . . 375
17.3.3 Resource Cost . . . 375
17.3.4 The Relationship of PathFinder to Lagrangian Relaxation . . . 376
17.3.5 Circuit Graph Extensions . . . 376
17.4 Parallel PathFinder . . . 377
17.5 Other Applications of the PathFinder Algorithm . . . 379
17.6 Summary . . . 379
References . . . 380

18 Retiming, Repipelining, and *C*-slow Retiming 383

18.1 Retiming: Concepts, Algorithm, and Restrictions . . . 384
18.2 Repipelining and *C*-slow Retiming . . . 388
18.2.1 Repipelining . . . 389
18.2.2 *C*-slow Retiming . . . 390
18.3 Implementations of Retiming . . . 393
18.4 Retiming on Fixed-frequency FPGAs . . . 394
18.5 *C*-slowing as Multi-threading . . . 395
18.6 Why Isn't Retiming Ubiquitous? . . . 398
References . . . 398

19 Configuration Bitstream Generation 401

19.1 The Bitstream . . . 403
19.2 Downloading Mechanisms . . . 406
19.3 Software to Generate Configuration Data . . . 407
19.4 Summary . . . 409
References . . . 409

20 Fast Compilation Techniques 411

20.1 Accelerating Classical Techniques . . . 414
20.1.1 Accelerating Simulated Annealing . . . 415
20.1.2 Accelerating PathFinder . . . 418
20.2 Alternative Algorithms . . . 422
20.2.1 Multiphase Solutions . . . 422
20.2.2 Incremental Place and Route . . . 425
20.3 Effect of Architecture . . . 427
20.4 Summary . . . 431
References . . . 432

Part IV: Application Development 435

21 Implementing Applications with FPGAs 439

21.1 Strengths and Weaknesses of FPGAs . . . 439
21.1.1 Time to Market . . . 439
21.1.2 Cost . . . 440
21.1.3 Development Time . . . 440
21.1.4 Power Consumption . . . 440
21.1.5 Debug and Verification . . . 440
21.1.6 FPGAs and Microprocessors . . . 441

21.2 Application Characteristics and Performance 441
21.2.1 Computational Characteristics and Performance 441
21.2.2 I/O and Performance 443
21.3 General Implementation Strategies for FPGA-based Systems 445
21.3.1 Configure-once 445
21.3.2 Runtime Reconfiguration 446
21.3.3 Summary of Implementation Issues 447
21.4 Implementing Arithmetic in FPGAs 448
21.4.1 Fixed-point Number Representation and Arithmetic 448
21.4.2 Floating-point Arithmetic 449
21.4.3 Block Floating Point 450
21.4.4 Constant Folding and Data-oriented Specialization 450
21.5 Summary 452
References 452

22 Instance-specific Design 455

22.1 Instance-specific Design 455
22.1.1 Taxonomy 456
22.1.2 Approaches 457
22.1.3 Examples of Instance-specific Designs 459
22.2 Partial Evaluation 462
22.2.1 Motivation 463
22.2.2 Process of Specialization 464
22.2.3 Partial Evaluation in Practice 464
22.2.4 Partial Evaluation of a Multiplier 466
22.2.5 Partial Evaluation at Runtime 470
22.2.6 FPGA-specific Concerns 471
22.3 Summary 473
References 473

23 Precision Analysis for Fixed-point Computation 475

23.1 Fixed-point Number System 475
23.1.1 Multiple-wordlength Paradigm 476
23.1.2 Optimization for Multiple Wordlength 478
23.2 Peak Value Estimation 478
23.2.1 Analytic Peak Estimation 479
23.2.2 Simulation-based Peak Estimation 484
23.2.3 Summary of Peak Estimation 485
23.3 Wordlength Optimization 485
23.3.1 Error Estimation and Area Models 485
23.3.2 Search Techniques 496
23.4 Summary 498
References 499

24 Distributed Arithmetic 503

24.1 Theory 503
24.2 DA Implementation 504
24.3 Mapping DA onto FPGAs 507
24.4 Improving DA Performance 508

24.5 An Application of DA on an FPGA 511
References 511

25 CORDIC Architectures for FPGA Computing 513

25.1 CORDIC Algorithm 514
25.1.1 Rotation Mode 514
25.1.2 Scaling Considerations 517
25.1.3 Vectoring Mode 519
25.1.4 Multiple Coordinate Systems and a Unified Description 520
25.1.5 Computational Accuracy 522
25.2 Architectural Design 526
25.3 FPGA Implementation of CORDIC Processors 527
25.3.1 Convergence 527
25.3.2 Folded CORDIC 528
25.3.3 Parallel Linear Array 530
25.3.4 Scaling Compensation 534
25.4 Summary 534
References 535

26 Hardware/Software Partitioning 539

26.1 The Trend Toward Automatic Partitioning 540
26.2 Partitioning of Sequential Programs 542
26.2.1 Granularity 545
26.2.2 Partition Evaluation 547
26.2.3 Alternative Region Implementations 549
26.2.4 Implementation Models 550
26.2.5 Exploration 552
26.3 Partitioning of Parallel Programs 557
26.3.1 Differences among Parallel Programming Models 557
26.4 Summary and Directions 558
References 559

Part V: Case Studies of FPGA Applications 561

27 SPIHT Image Compression 565

27.1 Background 565
27.2 SPIHT Algorithm 566
27.2.1 Wavelets and the Discrete Wavelet Transform 567
27.2.2 SPIHT Coding Engine 568
27.3 Design Considerations and Modifications 571
27.3.1 Discrete Wavelet Transform Architectures 571
27.3.2 Fixed-point Precision Analysis 575
27.3.3 Fixed Order SPIHT 578
27.4 Hardware Implementation 580
27.4.1 Target Hardware Platform 581
27.4.2 Design Overview 581
27.4.3 Discrete Wavelet Transform Phase 582
27.4.4 Maximum Magnitude Phase 583
27.4.5 The SPIHT Coding Phase 585

27.5 Design Results 587
27.6 Summary and Future Work 588
References 589

28 Automatic Target Recognition Systems on Reconfigurable Devices 591

28.1 Automatic Target Recognition Algorithms 592
28.1.1 Focus of Attention 592
28.1.2 Second-level Detection 592
28.2 Dynamically Reconfigurable Designs 594
28.2.1 Algorithm Modifications 594
28.2.2 Image Correlation Circuit 594
28.2.3 Performance Analysis 596
28.2.4 Template Partitioning 598
28.2.5 Implementation Method 599
28.3 Reconfigurable Static Design 600
28.3.1 Design-specific Parameters 601
28.3.2 Order of Correlation Tasks 601
28.3.3 Reconfigurable Image Correlator 602
28.3.4 Application-specific Computation Unit 603
28.4 ATR Implementations 604
28.4.1 A Dynamically Reconfigurable System 604
28.4.2 A Statically Reconfigurable System 606
28.4.3 Reconfigurable Computing Models 607
28.5 Summary 609
References 610

29 Boolean Satisfiability: Creating Solvers Optimized for Specific Problem Instances 613

29.1 Boolean Satisfiability Basics 613
29.1.1 Problem Formulation 613
29.1.2 SAT Applications 614
29.2 SAT-solving Algorithms 615
29.2.1 Basic Backtrack Algorithm 615
29.2.2 Improving the Backtrack Algorithm 617
29.3 A Reconfigurable SAT Solver Generated According to an SAT Instance 618
29.3.1 Problem Analysis 618
29.3.2 Implementing a Basic Backtrack Algorithm with Reconfigurable Hardware 619
29.3.3 Implementing an Improved Backtrack Algorithm with Reconfigurable Hardware 624
29.4 A Different Approach to Reduce Compilation Time and Improve Algorithm Efficiency 627
29.4.1 System Architecture 627
29.4.2 Performance 630
29.4.3 Implementation Issues 631
29.5 Discussion 633
References 635

30 Multi-FPGA Systems: Logic Emulation 637

30.1 Background . . . 637
30.2 Uses of Logic Emulation Systems . . . 639
30.3 Types of Logic Emulation Systems . . . 640
30.3.1 Single-FPGA Emulation . . . 640
30.3.2 Multi-FPGA Emulation . . . 641
30.3.3 Design-mapping Overview . . . 644
30.3.4 Multi-FPGA Partitioning and Placement Approaches . . . 645
30.3.5 Multi-FPGA Routing Approaches . . . 646
30.4 Issues Related to Contemporary Logic Emulation . . . 650
30.4.1 In-circuit Emulation . . . 650
30.4.2 Coverification . . . 650
30.4.3 Logic Analysis . . . 651
30.5 The Need for Fast FPGA Mapping . . . 652
30.6 Case Study: The VirtuaLogic VLE Emulation System . . . 653
30.6.1 The VirtuaLogic VLE Emulation System Structure . . . 653
30.6.2 The VirtuaLogic Emulation Software Flow . . . 654
30.6.3 Multiported Memory Mapping . . . 657
30.6.4 Design Mapping with Multiple Asynchronous Clocks . . . 657
30.6.5 Incremental Compilation of Designs . . . 661
30.6.6 VLE Interfaces for Coverification . . . 664
30.6.7 Parallel FPGA Compilation for the VLE System . . . 665
30.7 Future Trends . . . 666
30.8 Summary . . . 667
References . . . 668

31 The Implications of Floating Point for FPGAs 671

31.1 Why Is Floating Point Difficult? . . . 671
31.1.1 General Implementation Considerations . . . 673
31.1.2 Adder Implementation . . . 675
31.1.3 Multiplier Implementation . . . 677
31.2 Floating-point Application Case Studies . . . 679
31.2.1 Matrix Multiply . . . 679
31.2.2 Dot Product . . . 683
31.2.3 Fast Fourier Transform . . . 686
31.3 Summary . . . 692
References . . . 694

32 Finite Difference Time Domain: A Case Study Using FPGAs 697

32.1 The FDTD Method . . . 697
32.1.1 Background . . . 697
32.1.2 The FDTD Algorithm . . . 701
32.1.3 FDTD Applications . . . 703
32.1.4 The Advantages of FDTD on an FPGA . . . 705
32.2 FDTD Hardware Design Case Study . . . 707
32.2.1 The WildStar-II Pro FPGA Computing Board . . . 708
32.2.2 Data Analysis and Fixed-point Quantization . . . 709

32.2.3 Hardware Implementation 712
32.2.4 Performance Results 722
32.3 Summary 723
References 723

33 Evolvable FPGAs 725

33.1 The POE Model of Bioinspired Design Methodologies 725
33.2 Artificial Evolution 727
33.2.1 Genetic Algorithms 727
33.3 Evolvable Hardware 729
33.3.1 Genome Encoding 731
33.4 Evolvable Hardware: A Taxonomy 733
33.4.1 Extrinsic Evolution 733
33.4.2 Intrinsic Evolution 734
33.4.3 Complete Evolution 736
33.4.4 Open-ended Evolution 738
33.5 Evolvable Hardware Digital Platforms 739
33.5.1 Xilinx XC6200 Family 740
33.5.2 Evolution on Commercial FPGAs 741
33.5.3 Custom Evolvable FPGAs 743
33.6 Conclusions and Future Directions 745
References 747

34 Network Packet Processing in Reconfigurable Hardware 753

34.1 Networking with Reconfigurable Hardware 753
34.1.1 The Motivation for Building Networks with Reconfigurable Hardware 753
34.1.2 Hardware and Software for Packet Processing 754
34.1.3 Network Data Processing with FPGAs 755
34.1.4 Network Processing System Modularity 756
34.2 Network Protocol Processing 757
34.2.1 Internet Protocol Wrappers 758
34.2.2 TCP Wrappers 758
34.2.3 Payload-processing Modules 760
34.2.4 Payload Processing with Regular Expression Scanning . . 761
34.2.5 Payload Scanning with Bloom Filters 762
34.3 Intrusion Detection and Prevention 762
34.3.1 Worm and Virus Protection 763
34.3.2 An Integrated Header, Payload, and Queuing System . . . 764
34.3.3 Automated Worm Detection 766
34.4 Semantic Processing 767
34.4.1 Language Identification 767
34.4.2 Semantic Processing of TCP Data 768
34.5 Complete Networking System Issues 770
34.5.1 The Rack-mount Chassis Form Factor 770
34.5.2 Network Control and Configuration 771
34.5.3 A Reconfiguration Mechanism 772
34.5.4 Dynamic Hardware Plug-ins 773

34.5.5 Partial Bitfile Generation 773
34.5.6 Control Channel Security 774
34.6 Summary 775
References 776

35 Active Pages: Memory-centric Computation 779

35.1 Active Pages 779
35.1.1 DRAM Hardware Design 780
35.1.2 Hardware Interface 780
35.1.3 Programming Model 781
35.2 Performance Results 781
35.2.1 Speedup over Conventional Systems 782
35.2.2 Processor–Memory Nonoverlap 784
35.2.3 Summary 786
35.3 Algorithmic Complexity 786
35.3.1 Algorithms 787
35.3.2 Array-Insert 788
35.3.3 LCS (Two-dimensional Dynamic Programming) 791
35.3.4 Summary 794
35.4 Exploring Parallelism 794
35.4.1 Speedup over Conventional 795
35.4.2 Multiplexing Performance 796
35.4.3 Processor Width Performance 796
35.4.4 Processor Width versus Multiplexing 797
35.4.5 Summary 799
35.5 Defect Tolerance 799
35.6 Related Work 801
35.7 Summary 802
References 802

Part VI: Theoretical Underpinnings and Future Directions 805

36 Theoretical Underpinnings 807

36.1 General Computational Array Model 807
36.2 Implications of the General Model 809
36.2.1 Instruction Distribution 810
36.2.2 Instruction Storage 813
36.3 Induced Architectural Models 814
36.3.1 Fixed Instructions (FPGA) 815
36.3.2 Shared Instructions (SIMD Processors) 815
36.4 Modeling Architectural Space 816
36.4.1 Raw Density from Architecture 816
36.4.2 Efficiency 817
36.4.3 Caveats 825
36.5 Implications 826
36.5.1 Density of Computation versus Description 826
36.5.2 Historical Appropriateness 826
36.5.3 Reconfigurable Applications 827
References 828

37 Defect and Fault Tolerance 829

37.1 Defects and Faults . . . 830
37.2 Defect Tolerance . . . 830
37.2.1 Basic Idea . . . 830
37.2.2 Substitutable Resources . . . 832
37.2.3 Yield . . . 832
37.2.4 Defect Tolerance through Sparing . . . 835
37.2.5 Defect Tolerance with Matching . . . 840
37.3 Transient Fault Tolerance . . . 843
37.3.1 Feedforward Correction . . . 844
37.3.2 Rollback Error Recovery . . . 845
37.4 Lifetime Defects . . . 848
37.4.1 Detection . . . 848
37.4.2 Repair . . . 849
37.5 Configuration Upsets . . . 849
37.6 Outlook . . . 850
References . . . 850

38 Reconfigurable Computing and Nanoscale Architecture 853

38.1 Trends in Lithographic Scaling . . . 854
38.2 Bottom-up Technology . . . 855
38.2.1 Nanowires . . . 856
38.2.2 Nanowire Assembly . . . 857
38.2.3 Crosspoints . . . 857
38.3 Challenges . . . 858
38.4 Nanowire Circuits . . . 859
38.4.1 Wired-OR Diode Logic Array . . . 859
38.4.2 Restoration . . . 860
38.5 Statistical Assembly . . . 862
38.6 nanoPLA Architecture . . . 864
38.6.1 Basic Logic Block . . . 864
38.6.2 Interconnect Architecture . . . 867
38.6.3 Memories . . . 869
38.6.4 Defect Tolerance . . . 869
38.6.5 Design Mapping . . . 869
38.6.6 Density Benefits . . . 870
38.7 Nanoscale Design Alternatives . . . 870
38.7.1 Imprint Lithography . . . 870
38.7.2 Interfacing . . . 871
38.7.3 Restoration . . . 872
38.8 Summary . . . 872
References . . . 873

Index 877

LIST OF CONTRIBUTORS

Rajeevan Amirtharajah, Department of Electrical and Computer Engineering, University of California–Davis, Davis, California (Chapter 24)

Vaughn Betz, Altera Corporation, San Jose, California (Chapter 14)

Robert W. Brodersen, Department of Electrical Engineering and Computer Science, University of California–Berkeley, Berkeley, California (Chapter 8)

Timothy J. Callahan, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania (Chapter 7)

Eylon Caspi, Tabula, Inc., Santa Clara, California (Chapter 9)

Chen Chang, Department of Mathematics and Department of Electrical Engineering and Computer Sciences, University of California–Berkeley, Berkeley, California (Chapter 8)

Mark L. Chang, Electrical and Computer Engineering, Franklin W. Olin College of Engineering, Needham, Massachusetts (Chapter 1)

Wang Chen, Department of Electrical and Computer Engineering, Northeastern University, Boston, Massachusetts (Chapter 32)

Young H. Cho, Open Acceleration Systems Research, Chatsworth, California (Chapter 28)

Michael Chu, DRC Computer, Sunnyvale, California (Chapter 9)

Katherine Compton, Department of Electrical and Computer Engineering, University of Wisconsin–Madison, Madison, Wisconsin (Chapters 4 and 11)

Jason Cong, Department of Computer Science, California NanoSystems Institute, University of California–Los Angeles, Los Angeles, California (Chapter 13)

George A. Constantinides, Department of Electrical and Electronic Engineering, Imperial College, London, United Kingdom (Chapter 23)

André DeHon, Department of Electrical and Systems Engineering, University of Pennsylvania, Philadelphia, Pennsylvania (Chapters 5, 6, 7, 9, 11, 36, 37, and 38)

Chris Dick, Advanced Systems Technology Group, DSP Division of Xilinx, Inc., San Jose, California (Chapter 25)

Carl Ebeling, Department of Computer Science and Engineering, University of Washington, Seattle, Washington (Chapter 17)

Ken Eguro, Department of Electrical Engineering, University of Washington, Seattle, Washington (Chapter 20)

Diana Franklin, Computer Science Department, California Polytechnic State University, San Luis Obispo, California (Chapter 35)

PART I

RECONFIGURABLE COMPUTING HARDWARE

At a fundamental level, reconfigurable computing is the process of best exploiting the potential of reconfigurable hardware. Although a complete system must include compilation software and high-performance applications, the best place to begin to understand reconfigurable computing is at the chip level, as it is the abilities and limitations of chips that crucially influence all of a system's steps. However, the reverse is true as well—reconfigurable devices are designed primarily as a target for the applications that will be developed, and a chip that does not efficiently support important applications, or that cannot be effectively targeted by automatic design mapping flows, will not be successful.

Reconfigurable computing has been driven largely by the development of commodity field-programmable gate arrays (FPGAs). Standard FPGAs are somewhat of a mixed blessing for this field. On the one hand, they represent a source of commodity parts, offering cheap and fast programmable silicon on some of the most advanced fabrication processes available anywhere. On the other hand, they are not optimized for reconfigurable computing for the simple reason that the vast majority of FPGA customers use them as cheap, low-quality application-specific integrated circuits (ASICs) with rapid time to market. Thus, these devices are never quite what the reconfigurable computing user might want, but they are close enough. Chapter 1 covers commercial FPGA architectures in depth, providing an overview of the underlying technology for virtually all generally available reconfigurable computing systems.

Because FPGAs are not optimized toward reconfigurable computing, there have been many attempts to build better silicon devices for this community. Chapter 2 details many of them. The focus of the new architectures might be the inclusion of larger functional blocks to speed up important computations, tight connectivity to a host processor to set up a coprocessing model, fast reconfiguration features to reduce the time to change configurations, or other concepts. However, as of now, no such system is commercially viable, largely because

- The demand for reconfigurable computing chips is much smaller than that for the FPGA community as a whole, reducing economies of scale.
- FPGA manufacturers have access to cutting-edge fabrication processes, while reconfigurable computing chips typically are one to two process generations behind.

For these reasons, a reconfigurable computing chip is at a significant cost, performance, and electrical power-consumption disadvantage compared to a commodity FPGA. Thus, the architectural advantages of a reconfigurable computing-specific device must be huge to make up for the problems of less economies of scale and fabrication process lag. It seems likely that eventually a company with a reconfigurable computing-specific chip will be successful; however, so far there appears to have been only failures.

Although programmable chips are important, most reconfigurable computing users need more. A real system generally requires large memories, input/output (I/O) ports to hook to various data streams, microprocessors or microprocessor interfaces to coordinate operation, and mechanisms for configuring and reconfiguring the device. Chapter 3 considers such complete systems, chronicling the development of reconfigurable computing boards.

Chapters 1 through 3 present a good overview of most reconfigurable systems hardware, but one topic requires special consideration: the reconfiguration subsystems within devices. In the first FPGAs, configuration data was loaded slowly and sequentially, configuring the entire chip for a given computation. For glue logic and ASIC replacement, this was sufficient because FPGAs needed to be configured only once, at power-up; however, in many situations the device may need to be reconfigured more often. In the extreme, a single computation might be broken into multiple configurations, with the FPGA loading new configurations during the normal execution of that circuit. In this case, the speed of reconfiguration is important. Chapter 4 focuses on the configuration memory subsystems within an FPGA, considering the challenges of fast reconfiguration and showing some ways to greatly improve reconfiguration speed.

CHAPTER 1

DEVICE ARCHITECTURE

Mark L. Chang
Electrical and Computer Engineering
Franklin W. Olin College of Engineering

The best race car drivers understand how their cars work. The best architects know how carpenters, bricklayers, and electricians do their jobs. And the best programmers know how the hardware they are programming does computation. Knowing how your device works, "down to the metal," is essential for efficient utilization of available resources.

In this chapter, we take a look inside the package to discover the basic hardware elements that make up a typical field-programmable gate array (FPGA). We'll talk about how computation happens in an FPGA—from the blocks that do the computation to the interconnect that shuttles data from one place to another. We'll talk about how these building blocks fit together in terms of FPGA architecture. And, of course, because programmability (as well as reprogrammability) is part of what makes an FPGA so useful, we'll spend some time on that, too. Finally, we'll take an in-depth look at the architectures of some commercially available FPGAs in Section 1.5, Case Studies.

We won't be covering many of the research architectures from universities and industry—we'll save that for later. We also won't be talking much about how you successfully program these things to make them useful parts of a computational platform. That, too, is later in the book.

What you *will* learn is what's "under the hood" of a typical commercial FPGA so that you will become more comfortable using it as a platform for solving problems and performing computations. The first step in our journey starts with how computation in an FPGA is done.

1.1 LOGIC—THE COMPUTATIONAL FABRIC

Think of your typical desktop computer. Inside the case, among other things, are storage and communication devices (hard drives and network cards), memory, and, of course, the central processing unit, or CPU, where most of the computation happens. The FPGA plays a similar role in a reconfigurable computing platform, but we're going to break it down.

In very general terms, there are only two types of resources in an FPGA: *logic* and *interconnect*. Logic is where we do things like arithmetic, `1+1=2`, and logical functions, `if (ready) x=1 else x=0`. Interconnect is how we get data (like the

results of the previous computations) from one node of computation to another. Let's focus on logic first.

1.1.1 Logic Elements

From your digital logic and computer architecture background, you know that any computation can be represented as a Boolean equation (and in some cases as a Boolean equation where inputs are dependent on past results—don't worry, FPGAs can hold state, too). In turn, any Boolean equation can be expressed as a truth table. From these humble beginnings, we can build complex structures that can do arithmetic, such as adders and multipliers, as well as decision-making structures that can evaluate conditional statements, such as the classic if-then-else. Combining these, we can describe elaborate algorithms *simply by using truth tables*.

From this basic observation of digital logic, we see the truth table as the computational heart of the FPGA. More specifically, one hardware element that can easily implement a truth table is the lookup table, or LUT. From a circuit implementation perspective, a LUT can be formed simply from an N:1 (N-to-one) multiplexer and an N-bit memory. From the perspective of our previous discussion, a LUT simply enumerates a truth table. Therefore, using LUTs gives an FPGA the generality to implement arbitrary digital logic. Figure 1.1 shows a typical N-input lookup table that we might find in today's FPGAs. In fact, almost all commercial FPGAs have settled on the LUT as their basic building block.

The LUT can compute any function of N inputs by simply programming the lookup table with the truth table of the function we want to implement. As shown in the figure, if we wanted to implement a 3-input exclusive-or (XOR) function with our 3-input LUT (often referred to as a 3-LUT), we would assign values to the lookup table memory such that the pattern of select bits chooses the correct row's "answer." Thus, every "row" would yield a result of 0 except in the four cases where the XOR of the three select lines yields 1.

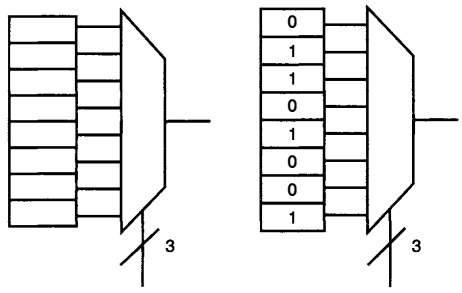


FIGURE 1.1 ■ A 3-LUT schematic (a) and the corresponding 3-LUT symbol and truth table (b) for a logical XOR.

Of course, more complicated functions, and functions of a larger number of inputs, can be implemented by aggregating several lookup tables together. For example, one can organize a single 3-LUT into an 8×1 ROM, and if the values of the lookup table are reprogrammable, an 8×1 RAM. But the basic building block, the lookup table, remains the same.

Although the LUT has more or less been chosen as the smallest computational unit in commercially available FPGAs, the size of the lookup table in each logic block has been widely investigated [1]. On the one hand, larger lookup tables would allow for more complex logic to be performed per logic block, thus reducing the wiring delay between blocks as fewer blocks would be needed. However, the penalty paid would be slower LUTs, because of the requirement of larger multiplexers, and an increased chance of waste if not all of the functionality of the larger LUTs were to be used. On the other hand, smaller lookup tables may require a design to consume a larger number of logic blocks, thus increasing wiring delay between blocks while reducing per–logic block delay.

Current empirical studies have shown that the 4-LUT structure makes the best trade-off between area and delay for a wide range of benchmark circuits. Of course, as FPGA computing evolves into wider arenas, this result may need to be revisited. In fact, as of this writing, Xilinx has released the Virtex-5 SRAM-based FPGA with a 6-LUT architecture.

The question of the number of LUTs per logic block has also been investigated [2], with empirical evidence suggesting that grouping more than one 4-LUT into a single logic block may improve area and delay. Many current commercial FPGAs incorporate a number of 4-LUTs into each logic block to take advantage of this observation.

Investigations into both LUT size and number of LUTs per block begin to address the larger question of computational *granularity* in an FPGA. On one end of the spectrum, the rather simple structure of a small lookup table (e.g., 2-LUT) represents *fine-grained* computational capability. Toward the other end, *coarse-grained*, one can envision larger computational blocks, such as full 8-bit arithmetic logic units (ALUs), more typical of CPUs. As in the case of lookup table sizing, finer-grained blocks may be more adept at bit-level manipulations and arithmetic, but require combining several to implement larger pieces of logic. Contrast that with coarser-grained blocks, which may be more optimal for datapath-oriented computations that work with standard "word" sizes (8/16/32 bits) but are wasteful when implementing very simple logical operations. Current industry practice has been to strike a balance in granularity by using rather fine-grained 4-LUT architectures and augmenting them with coarser-grained heterogeneous elements, such as multipliers, as described in the Extended Logic Elements section later in this chapter.

Now that we have chosen the logic block, we must ask ourselves if this is sufficient to implement all of the functionality we want in our FPGA. Indeed, it is not. With just LUTs, there is no way for an FPGA to maintain any sense of state, and therefore we are prohibited from implementing any form of sequential, or state-holding, logic. To remedy this situation, we will add a simple single-bit storage element in our base logic block in the form of a D flip-flop.

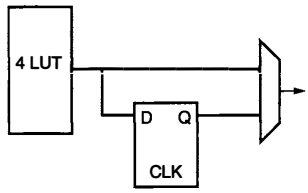


FIGURE 1.2 ■ A simple lookup table logic block.

Now our logic block looks something like Figure 1.2. The output multiplexer selects a result either from the function generated by the lookup table or from the stored bit in the D flip-flop. In reality, this logic block bears a very close resemblance to those in some commercial FPGAs.

1.1.2 Programmability

Looking at our logic block in Figure 1.2, it is a simple task to identify all the programmable points. These include the contents of the 4-LUT, the select signal for the output multiplexer, and the initial state of the D flip-flop. Most current commercial FPGAs use volatile static-RAM (SRAM) bits connected to configuration points to configure the FPGA. Thus, simply writing a value to each configuration bit sets the configuration of the entire FPGA.

In our logic block, the 4-LUT would be made up of 16 SRAM bits, one per output; the multiplexer would use a single SRAM bit; and the D flip-flop initialization value could also be held in a single SRAM bit. How these SRAM bits are initialized in the context of the rest of the FPGA will be the subject of later sections.

1.2 THE ARRAY AND INTERCONNECT

With the LUT and D flip-flop, we begin to define what is commonly known as the *logic block*, or *function block*, of an FPGA. Now that we have an understanding of how computation is performed in an FPGA at the single logic block level, we turn our focus to how these computation blocks can be tiled and connected together to form the fabric that is our FPGA.

Current popular FPGAs implement what is often called *island-style* architecture. As shown in Figure 1.3, this design has logic blocks tiled in a two-dimensional array and interconnected in some fashion. The logic blocks form the islands and "float" in a sea of interconnect.

With this array architecture, computations are performed spatially in the fabric of the FPGA. Large computations are broken into 4-LUT-sized pieces and mapped into physical logic blocks in the array. The interconnect is configured to route signals between logic blocks appropriately. With enough logic blocks, we can make our FPGAs perform any kind of computation we desire.

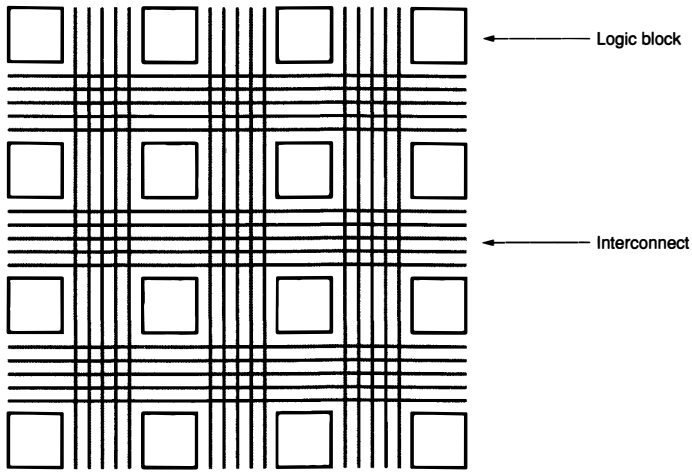


FIGURE 1.3 ■ The island-style FPGA architecture. The interconnect shown here is not representative of structures actually used.

1.2.1 Interconnect Structures

Figure 1.3 does not tell the whole story. The interconnect structure shown is not representative of any structures used in actual FPGAs, but is more of a cartoon placeholder. This section introduces the interconnect structures present in many of today's FPGAs, first by considering a small area of interconnection and then expanding out to understand the need for different styles of interconnect. We start with the simplest case of nearest-neighbor communication.

Nearest neighbor

Nearest-neighbor communication is as simple as it sounds. Looking at a 2×2 array of logic blocks in Figure 1.4, one can see that the only needs in this neighborhood are input and output connections in each direction: north, south, east, and west. This allows each logic block to communicate directly with each of its immediate neighbors.

Figure 1.4 is an example of one of the simplest routing architectures possible. While it may seem nearly degenerate, it has been used in some (now obsolete) commercial FPGAs. Of course, although this is a simple solution, this structure suffers from severe delay and connectivity issues. Imagine, instead of a 2×2 array, a 1024×1024 array. With only nearest-neighbor connectivity, the delay scales linearly with distance because the signal must go through many cells (and many switches) to reach its final destination.

From a connectivity standpoint, without the ability to bypass logic blocks in the routing structure, all routes that are more than a single hop away require

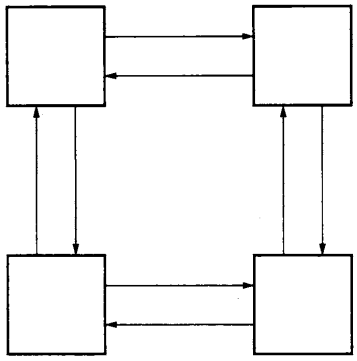


FIGURE 1.4 ■ Nearest-neighbor connectivity.

traversing a logic block. With only one bidirectional pair in each direction, this limits the number of logic block signals that may cross. Signals that are passing through must not overlap signals that are being actively consumed and produced.

Because of these limitations, the nearest-neighbor structure is rarely used *exclusively*, but it is almost always available in current FPGAs, often augmented with some of the techniques that follow.

Segmented

As we add complexity, we begin to move away from the pure logic block architecture that we've developed thus far. Most current FPGA architectures look less like Figure 1.3 and more like Figure 1.5.

In Figure 1.5 we introduce the connection block and the switch box. Here the routing structure is more generic and meshlike. The logic block accesses nearby communication resources through the connection block, which connects logic block input and output terminals to routing resources through programmable switches, or multiplexers. The connection block (detailed in Figure 1.6) allows logic block inputs and outputs to be assigned to arbitrary horizontal and vertical tracks, increasing routing flexibility.

The switch block appears where horizontal and vertical routing tracks converge as shown in Figure 1.7. In the most general sense, it is simply a matrix of programmable switches that allow a signal on a track to connect to another track. Depending on the design of the switch block, this connection could be, for example, to turn the corner in either direction or to continue straight. The design of switch blocks is an entire area of research by itself and has produced many varied designs that exhibit varying degrees of connectivity and efficiency [3–5]. A detailed discussion of this research is beyond the scope of this book.

With this slightly modified architecture, the concept of a segmented interconnect becomes more clear. Nearest-neighbor routing can still be accomplished, albeit through a pair of connect blocks and a switch block. However, for

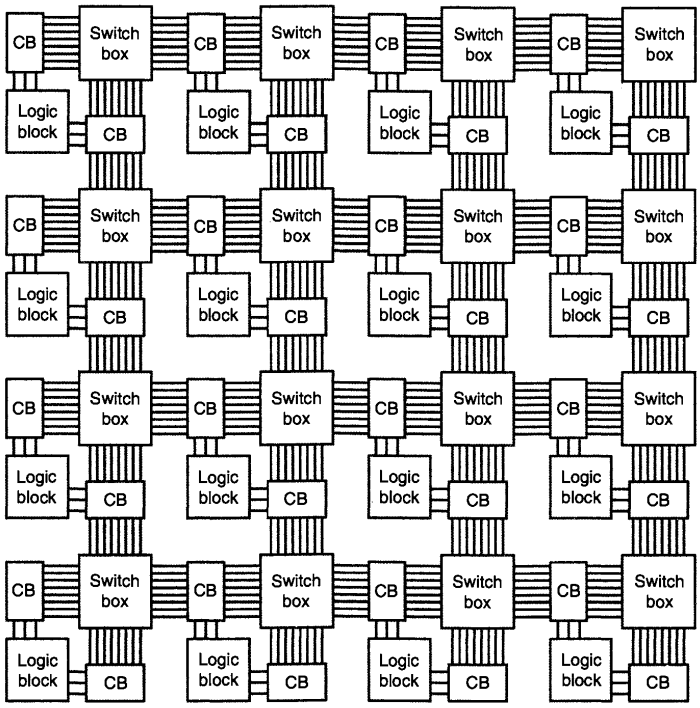


FIGURE 1.5 ■ An island-style architecture with connect blocks and switch boxes to support more complex routing structures. (The difference in relative sizes of the blocks is for visual differentiation.)

signals that need to travel longer distances, individual segments can be switched together in a switch block to connect distant logic blocks together. Think of it as a way to emulate long signal paths that can span arbitrary distances. The result is a long wire that actually comprises shorter "segments."

This interconnect architecture alone does not radically improve on the delay characteristics of the nearest-neighbor interconnect structure. However, the introduction of connection blocks and switch boxes separates the interconnect from the logic, allowing long-distance routing to be accomplished without consuming logic block resources.

To improve on our structure, we introduce longer-length wires. For instance, consider a wire that spans one logic block as being of length-1 (L1). In some segmented routing architectures, longer wires may be present to allow signals to travel greater distances more efficiently. These segments may be

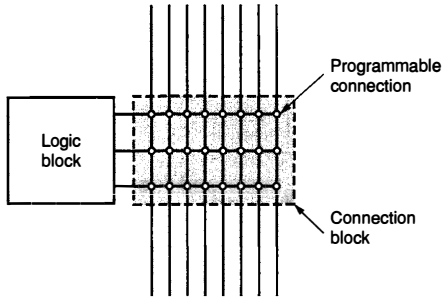


FIGURE 1.6 ■ Detail of a connection block.

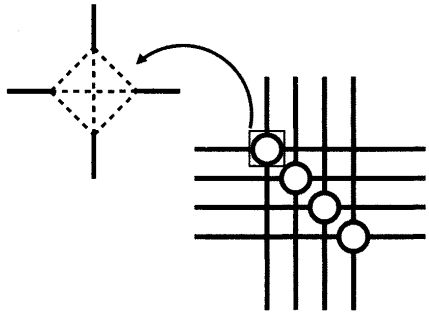


FIGURE 1.7 ■ An example of a common switch block architecture.

length-4 (L4), length-8 (L8), and so on. The switch blocks (and erhaps more embedded switches) become points where signals can switch from shorter to longer segments. This feature allows signal delay to be less than $O(N)$ when covering a distance of N logic blocks by reducing the number of intermediate switches in the signal path.

Figure 1.8 illustrates augmenting the single-segment interconnect with two additional lengths: direct-connect between logic blocks and length-2 (L2) lines. The direct-connect lines leave general routing resources free for other uses, and L2 lines allow signals to travel longer distances for roughly the same amount of switch delay. This interconnect architecture closely matches that of the Xilinx XC4000 series of commercial FPGAs.

Hierarchical

A slightly different approach to reducing the delay of long wires uses a hierarchical approach. Consider the structure in Figure 1.9. At the lowest level of hierarchy, 2×2 arrays of logic blocks are grouped together as a single cluster.

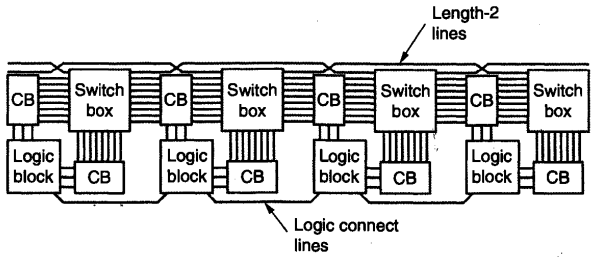


FIGURE 1.8 ■ Local (direct) connections and L2 connections augmenting a switched interconnect.

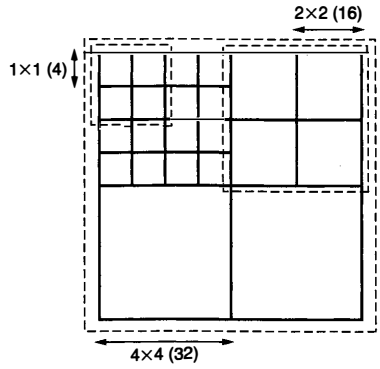


FIGURE 1.9 ■ Hierarchical routing used by long wires to connect clusters of logic blocks.

Within this block, local, nearest-neighbor routing is all that is available. In turn, a 2×2 cluster of these clusters is formed that encompasses 16 logic blocks. At this level of hierarchy, longer wires at the boundary of the smaller, 2×2 clusters, connect each cluster of four logic blocks to the other clusters in the higher-level grouping. This is repeated in higher levels of hierarchy, with larger clusters and longer wires.

The pattern of interconnect just described exploits the assumption that a well-designed (and well-placed) circuit has mostly local connections and only a limited number of connections that need to travel long distances. By providing fewer resources at the higher levels of hierarchy, this interconnect architecture remains area-efficient while preserving some long-length wires to minimize the delay of signals that need to cross large distances.

As in the segmented architecture, the connection points that connect one level of routing hierarchy to another can be anywhere in the interconnect structure. New points in the existing switch blocks may be created, or completely independent

switching sites elsewhere in the interconnect can be created specifically for the purpose of moving between hierarchy levels.

1.2.2 Programmability

As with the logic blocks in a typical commercial FPGA, each switch point in the interconnect structure is programmable. Within the connection block, programmable multiplexers select which routing track each logic block's input and output terminals map to; in the switch block, the junction between vertical and horizontal routing tracks is switched through a programmable switch; and, finally, switching between routing tracks of different segment lengths or hierarchy levels is accomplished, again through programmable switches.

For all of these programmable points, as in the logic block, modern FPGAs use SRAM bits to hold the user-defined configuration values. More discussion of these configuration bits comes later in this chapter.

1.2.3 Summary

Programmable routing resources are the natural counterpart to the logic resources in an FPGA. Where the logic performs the arithmetic and logical computations, the interconnection fabric takes the results output from logic blocks and routes them as inputs to other logic blocks. By tiling logic blocks together and connecting them through a series of programmable interconnects as described here, an FPGA can implement complex digital circuits. The true nature of *spatial computing* is realized by spreading the computation across the physical area of an FPGA.

Today's commercial FPGAs typically use bits of each of these interconnect architectures to provide a smooth and flexible set of routing resources. In actual implementation, segmentation and hierarchy may not always exhibit the logarithmic scaling seen in our examples. In modern FPGAs, the silicon area consumed by interconnect greatly dominates the area dedicated to logic. Anecdotally, 90 percent of the available silicon is interconnect whereas only 10 percent is logic. With this imbalance, it is clear that interconnect architecture is increasingly important, especially from a delay perspective.

1.3 EXTENDING LOGIC

With a logic block like the one shown in Figure 1.2, tiled in a two-dimensional array with a supporting interconnect structure, we can implement any combinational and sequential logic. Our only constraint is area in terms of the number of available logic blocks. While this is comprehensive, it is far from optimal. In this section, we investigate how FPGA architects have augmented this simple design to increase performance.

1.3.1 Extended Logic Elements

Modern FPGA interconnect architectures have matured to include much more than simple nearest-neighbor connectivity to give increased performance for

common applications. Likewise, the basic logic elements have been augmented to increase performance for common operations such as arithmetic functions and data storage.

Fast carry chain

One fundamental operation that the FPGA is likely to perform is an addition. From the basic logic block, it is apparent that we can implement a full-adder structure with two logic blocks given at least a 3-LUT. One logic block is configured to compute the sum, and one is configured to compute the carry. Cascading N pairs of logic blocks together will yield a simple N-bit full adder.

As you may already know from digital arithmetic, the critical path of this type of addition comes not from the computation of the sum bits but rather from the rippling of the carry signal from lower-order bits to higher-order bits (see Figure 1.10). This path starts with the low-order primary inputs, goes through the logic block, out into the interconnect, into the adjacent logic block, and so on. Delay is accumulated at every switch point along the way.

One clever way to increase speed is to shortcut the carry chain between adjacent logic blocks. We can accomplish this by providing a dedicated, minimally switched path from the output of the logic block computing the carry signal to the adjacent higher-order logic block pair. This carry chain will not need to be routed on the general interconnect network. By adding a minimal amount of overhead (wires), we dramatically speed up the addition operation.

This feature does force some constraints on the spatial layout of a multibit addition. If, for instance, the dedicated fast carry chain only goes vertically, along columns of logic blocks, all additions must be oriented along the carry chain to take advantage of this dedicated resource. Additionally, to save switching area, the dedicated carry chain may not be a bidirectional path, which further restricts the physical layout to be oriented vertically and dictates the order of the bits relative to one another. The fast carry-chain of the Xilinx XC4000E is shown in Figure 1.11. Note that the bidirectional fast carry-chain wires are arranged along the columns while the horizontal lines are unidirectional. This allows large adder structures to be placed in a zig-zag pattern in the array and still make use of the dedicated carry-chain interconnect.

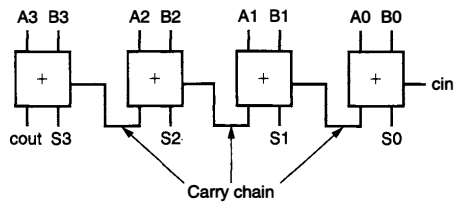


FIGURE 1.10 ■ A simple 4-bit full adder.

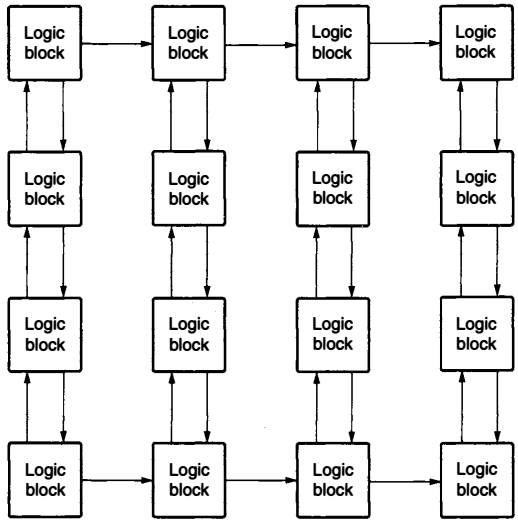


FIGURE 1.11 ■ The Xilinx XC4000E fast carry chain. (*Source:* Adapted from [6], Figure 11, p. 6-18.)

The fast carry-chain logic is now commonplace in commercial FPGAs, with the physical design constraints at this point completely abstracted away by the tools provided by manufacturers. The success of this optimization relies on the toolset's ability to identify additions in the designer's circuit description and then use the dedicated logic. With today's tools, this kind of optimization is nearly transparent to the end user.

Multipliers

If addition is commonplace in algorithms, multiplication is certainly not rare. Several implementations are available if we wish to use general logic block resources to build our multipliers. From the area-efficient iterative shift-accumulate method to the area-consumptive array multiplier, we can use logic blocks to either compute additions or store intermediate values. While we can certainly implement a multiplication, we can do so only with a large delay penalty, or a large logic block footprint, depending on our implementation. In essence, our logic blocks *aren't very efficient* at performing a multiplication.

Instead of doing it with logic blocks, why not build *real* multipliers outside, but still connected to, the general FPGA fabric? Then, instead of inefficiently using simple LUTs to implement a multiply, we can route the values that need to be multiplied to actual multipliers implemented in silicon. How does this save space and time? Recall that FPGAs trade speed and power for configurability when compared to their ASIC (application-specific integrated circuit) counterparts. If you asked a VLSI designer to implement a fast multiplier out of transistors

any way she wanted, it would take up far less silicon area, be much faster, and consume less power than we could ever manage using LUTs.

The result is that, for a small price in silicon area, we can offload the otherwise area-prohibitive multiplication onto dedicated hardware that does it much better. Of course, just like fast carry chains, multipliers impose important design considerations and physical constraints, but we add one more option for computation to our palette of operations. It is now just a matter of good design and good tools to make an efficient design. Like fast carry chains, multipliers are commonplace in modern FPGAs.

RAM

Another area that has seen some customization beyond the general FPGA fabric is in the area of on-chip data storage. While logic blocks can individually provide a few bits of storage via the lookup table structure—and, in aggregate, many bits—they are far from an efficient use of FPGA resources. Like the fast carry chain and the "hard" multiplier, FPGA architectures have given their users generous amounts of on-chip RAM that can be accessed from the general FPGA fabric.

Static RAM cells are extremely small and, when physically distributed throughout the FPGA, can be very useful for many algorithms. By grouping many static RAM cells into banks of memory, designers can implement large ROMs for extremely fast lookup table computations and constant-coefficient operations, and large RAMs for buffering, queuing, and basic scratch use—all with the convenience of a simple clocking strategy and the speed gained by avoiding off-chip communication to an external memory. Today's FPGAs provide anywhere from kilobits to megabits of dedicated RAM.

Processor blocks

Tying all these blocks together, most commercial FPGAs now offer entire dedicated processors in the FPGA, sometimes even more than one. In a general sense, FPGAs are extremely efficient at implementing raw computational pipelines, exploiting nonstandard bit widths, and providing data and functional parallelism. The inclusion of dedicated CPUs recognizes the fact that algorithm flows that are very procedural and contain a high degree of branching do not lend themselves readily to acceleration using FPGAs.

Entire CPU blocks can now be found in high-end FPGA devices. At the time of this writing, these CPUs are on the scale of 300 MHz PowerPC devices, complete, without floating-point units. They are capable of running an entire embedded operating system, and some are even able to reprogram the FPGA fabric around them.

The CPU cores are not nearly as easily exploited as the carry chains, multipliers, and on-chip RAMs, but they represent a distinct shift toward making FPGAs more "platform"-oriented. With a traditional CPU on board (and perhaps up to four), a single FPGA can serve nearly as an entire "system-on-a-chip"—the holy grail of system integrators and embedded device manufacturers. With standard programming languages and toolchains available to developers, an entire project might indeed be implemented with a single-chip solution, dramatically reducing cost and time to market.

1.3.2 Summary

In the end, modern commercially available FPGAs provide a rich variety of basic, and not so basic, computational building blocks. With much more than simple lookup tables, the task for the FPGA architect is to decide in what proportion to provide these resources and how they should be connected. The task of the hardware designer is then to fully understand the capabilities of the target FPGAs to create designs that exploit their potential.

The common thread among these extended logical elements is that they provide critical functionality that cannot be implemented very efficiently in the general FPGA fabric. As much as the technology drives FPGA architectures, applications provide a much needed push. If multiplies were rare, it wouldn't make sense to waste silicon space on a "hard" multiplier. As FPGAs become more heterogeneous in nature, and become useful computational platforms in new application domains, we can expect to see even more varied blocks in the next generation of devices.

1.4 CONFIGURATION

One of the defining features of an FPGA is its ability to act as "blank hardware" for the end user. Providing more performance than pure software implementations on general-purpose processors, and more flexibility than a fixed-function ASIC solution, relies on the FPGA being a reconfigurable device. In this section, we will discuss the different approaches and technologies used to provide programmability in an FPGA.

Each configurable element in an FPGA requires 1 bit of storage to maintain a user-defined configuration. For a simple LUT-based FPGA, these programmable locations generally include the contents of the logic block and the connectivity of the routing fabric. Configuration of the FPGA is accomplished through programming the storage bits connected to these programmable locations according to user definitions. For the lookup tables, this translates into filling it with 1s and 0s. For the routing fabric, programming enables and disables switches along wiring paths.

The configuration can be thought of as a flat binary file whose contents map, bit for bit, to the programmable bits in the FPGA. This *bitstream* is generated by the vendor-specific tools after a hardware design is finalized. While its exact format is generally not publicly known, the larger the FPGA, the larger the bitstream becomes.

Of course, there are many known methods for storing a single bit of binary information. We discuss the most popular methods used for FPGAs next.

1.4.1 SRAM

As discussed in previous sections, the most widely used method for storing configuration information in commercially available FPGAs is volatile static RAM, or SRAM. This method has been made popular because it provides fast and infinite reconfiguration in a well-known technology.

Drawbacks to SRAM come in the form of power consumption and data volatility. Compared to the other technologies described in this section, the SRAM cell is large (6–12 transistors) and dissipates significant static power because of leakage current. Another significant drawback is that SRAM does not maintain its contents without power, which means that at power-up the FPGA is not configured and must be programmed using off-chip logic and storage. This can be accomplished with a nonvolatile memory store to hold the configuration and a micro-controller to perform the programming procedure. While this may seem to be a trivial task, it adds to the component count and complexity of a design and prevents the SRAM-based FPGA from being a truly single-chip solution.

1.4.2 Flash Memory

Although less popular than SRAM, several families of devices use Flash memory to hold configuration information. Flash memory is different from SRAM in that it is nonvolatile and can only be written a finite number of times.

The nonvolatility of Flash memory means that the data written to it remains when power is removed. In contrast with SRAM-based FPGAs, the FPGA remains configured with user-defined logic even through power cycles and does not require extra storage or hardware to program at boot-up. In essence, a Flash-based FPGA can be ready immediately.

A Flash memory cell can also be made with fewer transistors compared to an SRAM cell. This design can yield lower static power consumption as there are fewer transistors to contribute to leakage current.

Drawbacks to using Flash memory to store FPGA configuration information stem from the techniques necessary to write to it. As mentioned, Flash memory has a limited write cycle lifetime and often has slower write speeds than SRAM. The number of write cycles varies by technology, but is typically hundreds of thousands to millions. Additionally, most Flash write techniques require higher voltages compared to normal circuits; they require additional off-chip circuitry or structures such as charge pumps on-chip to be able to perform a Flash write.

1.4.3 Antifuse

A third approach to achieving programmability is antifuse technology. Antifuse, as its name suggests, is a metal-based link that behaves the opposite of a fuse. The antifuse link is normally open (i.e., unconnected). A programming procedure that involves either a high-current programmer or a laser melts the link to form an electrical connection across it—in essence, creating a wire or a short-circuit between the antifuse endpoints.

Antifuse has several advantages and one clear disadvantage, which is that it is not reprogrammable. Once a link is fused, it has undergone a physical transformation that cannot be reversed. FPGAs based on this technology are generally considered one-time programmable (OTP). This severely limits their flexibility in terms of reconfigurable computing and nearly eliminates this technology for use in prototyping environments.

However, there are some distinct advantages to using antifuse in an FPGA platform. First, the antifuse link can be made very small, compared to the large multi-transistor SRAM cell, and does not require any transistors. This results in very low propagation delays across links and zero static power consumption, as there is no longer any transistor leakage current. Antifuse links are also not susceptible to high-energy radiation particles that induce errors known as single-event upsets, making them more likely candidates for space and military applications.

1.4.4 Summary

There are several well-known methods for storing user-defined configuration data in an FPGA. We have reviewed the three most common in this section. Each has its strengths and weaknesses, and all can be found in current commercial FPGA products.

Regardless of the technology used to store or convey configuration data, the idea remains the same. From vendor-specific tools, a device-specific programming bitstream is created and used either to program an SRAM or Flash memory, or to describe the pattern of antifuse links to be used. In the end, the user-defined configuration is reflected in the FPGA, bringing to reality part of the vision of reconfigurable computing.

1.5 CASE STUDIES

If you've read everything thus far, the FPGA should no longer seem like a magical computational black box. In fact, you should have a good grasp of the components that make up modern commercial FPGAs and how they are put together. In this section, we'll take it one step further and solidify the abstractions by taking a look at two real commercial architectures—the Altera Stratix and the Xilinx Virtex-II Pro—and linking the ideas introduced earlier in this chapter with concrete industry implementations.

Although these devices represent near-current technologies, having been introduced in 2002, they are not the latest generation of devices from their respective manufacturers. The reason for choosing them over more cutting-edge examples is in part due to the level of documentation available at the time of this writing. As is often the case, detailed architecture information is not available as soon as a product is released and may never be available depending on the manufacturer.

Finally, the devices discussed here are much more complex than we have space to describe. The myriad ways modern devices can be used to perform computation and the countless hardware and software features that allow you to create powerful and efficient designs are all part of a larger, more advanced dialog. So if something seems particularly interesting, we encourage you to grab a copy of the device handbook(s) and dig a little deeper.

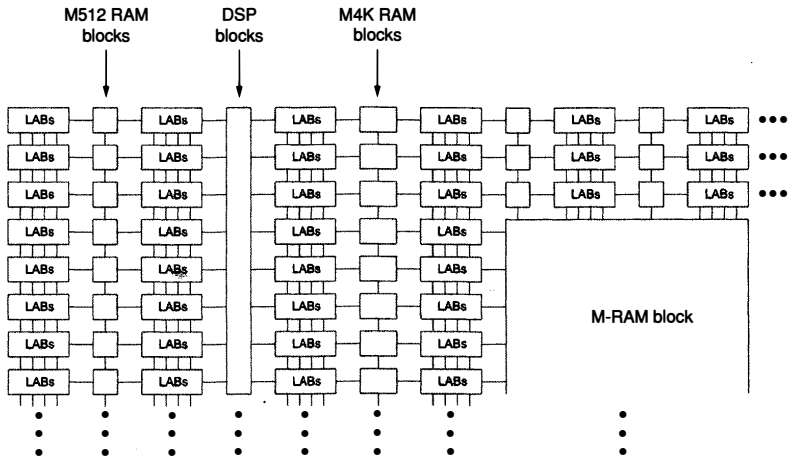


FIGURE 1.12 ■ Altera Stratix block diagram. (*Source:* Adapted from [7], Chapter 2, p. 2-2.)

1.5.1 Altera Stratix

We begin by taking a look at the Altera Stratix FPGA. Much of the information presented here is adapted from the July 2005 edition of the *Altera Stratix Device Handbook* (available online at *http://www.altera.com*).

The Stratix is an SRAM-based island-style FPGA containing many heterogeneous computational elements. The basic logical tile is the logic array block (LAB), which consists of 10 logic elements (LEs). The LABs are tiled across the device in rows and columns with a multilevel interconnect bringing together logic, memory, and other resources. Memory is provided through TriMatrix memory structures, which consist of three memory block sizes—M512, M4K, and M-RAM—each with its own unique properties. Additional computational resources are provided in DSP blocks, which can efficiently perform multiplication and accumulation. These resources are shown in a high-level block diagram in Figure 1.12.

Logic architecture

The smallest logical block in the array is the LE, shown in Figure 1.13. The general architecture of the LE is very similar to the structure that we introduced earlier—a single 4-LUT function generator and a programmable register as a state-holding element. In the Altera LE, you can see additional components to facilitate driving the interconnect (*right* side of Figure 1.12), setting and clearing the programmable register, choosing from several programmable clocks, and propagating the carry chain.

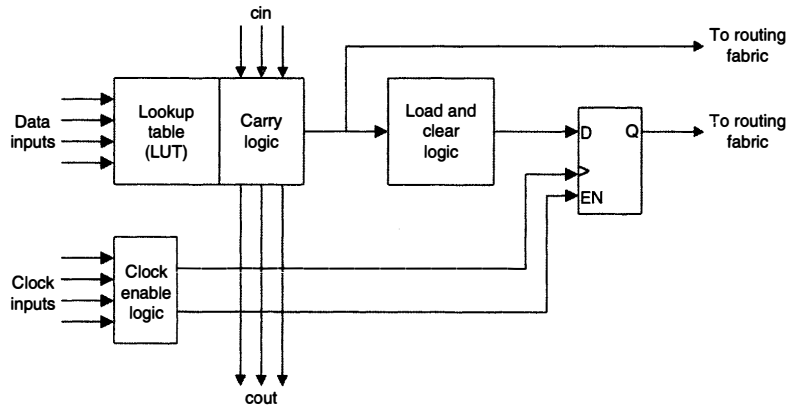


FIGURE 1.13 ■ Simplified Altera Stratix logic element. (*Source:* Adapted from [7], Chapter 2, p. 2-5.)

Because the LEs are simple structures that may appear tens of thousands of times in a single device, Altera groups them into LABs. The LAB is then the basic structure that is tiled into an array and connected via the routing structure. Each LAB consists of 10 LEs, all LE carry chains, LAB-wide control signals, and several local interconnection lines. In the largest device, the EP1S80, there are 101 LAB rows and 91 LAB columns, yielding a total of 79,040 LEs. This is fewer than would be expected given the number of rows and columns because of the presence of the TriMatrix memory structures and DSP blocks embedded in the array.

As shown in Figure 1.14, the LAB structure is dominated, at least conceptually, by interconnect. The local interconnect allows LEs in the same LAB to send signals to one another without using the general interconnect. Neighboring LABs, RAM blocks, and DSP blocks can also drive the local interconnect through direct links. Finally, the general interconnect (both horizontal and vertical channels) can drive the local interconnect. This high degree of connectivity is the lowest level of a rich, multilevel routing fabric.

The Stratix has three types of memory blocks—M512, M4K, and M-RAM—collectively dubbed TriMatrix memory. The largest distinction between these blocks is their size and number in a given device. Generally speaking, they can be configured in a number of ways, including single-port RAM, dual-port RAM, shift-register, FIFO, and ROM table. These memories can optionally include parity bits and have registered inputs and outputs.

The M512 RAM block is nominally organized as a 32×18-bit memory; the M4K RAM as a 128×36-bit memory; and the M-RAM as a $4K \times 144$-bit memory. Additionally, each block can be configured for a variety of widths depending on the needs of the user. The different-sized memories throughout the array provide

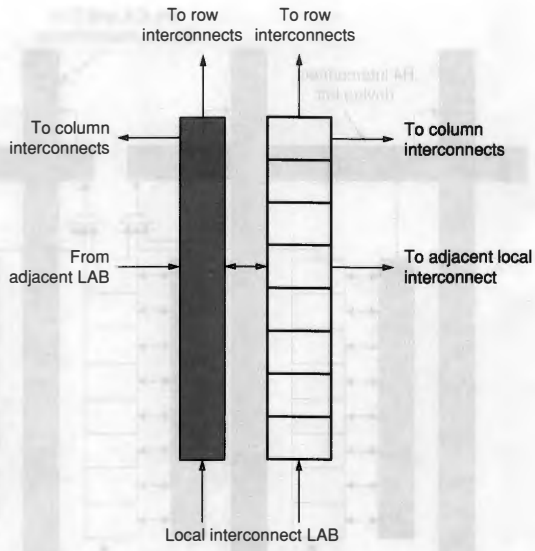


FIGURE 1.14 ■ Simplified Altera Stratix LAB structure. (*Source:* Adapted from [8], Chapter 2, p. 2-4.)

an efficient mapping of variable-sized memory designs to the device. In total, on the EP1S80 there are over 7 million memory bits available for use, divided into 767 M512 blocks, 364 M4K blocks, and 9 M-RAM blocks.

The final element of logic present in the Altera Stratix is the DSP block. Each device has two columns of DSP blocks that are designed to help implement DSP-type functions, such as finite-impulse response (FIR) and infinite-impulse response (IIR) filters and fast Fourier transforms (FFT), without using the general logic resources of the LEs. The common computational function required in these operations is often a multiplication and an accumulation. Each DSP block can be configured by the user to support a single 36×36-bit multiplication, four 18×18-bit multiplications, or eight 9×9-bit multiplications, in addition to an optional accumulation phase. In the EP1S80, there are 22 total DSP blocks.

Routing architecture

The Altera Stratix provides an interconnect system dubbed MultiTrack that connects all the elements just discussed using routing lines of varying fixed lengths. Along the row (horizontal) dimension, the routing resources include direct connections left and right between blocks (LABs, RAMs, and DSP) and interconnects of lengths 4, 8, and 24 that traverse either 4, 8, or 24 blocks left and right, respectively. A detailed depiction of an R4 interconnect at a single

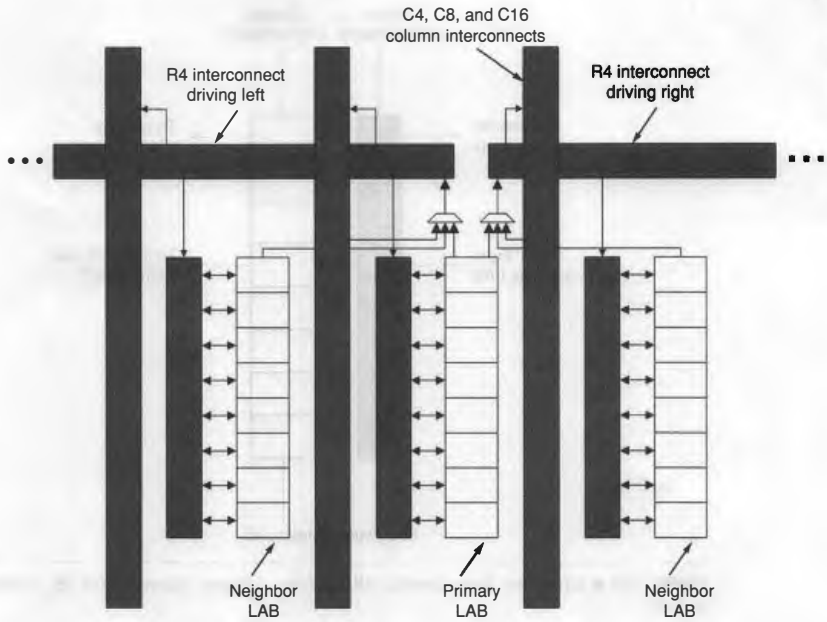


FIGURE 1.15 ■ Simplified Altera Stratix MultiTrack interconnect. (*Source:* Adapted from [7], Chapter 2, p. 2-14.)

LAB is shown in Figure 1.15. The R4 interconnect shown spans 4 blocks, left to right. The relative sizing of blocks in the Stratix allows the R4 interconnect to span four LABs; three LABs and one M512 RAM; two LABs and one M4K RAM; or two LABs and one DSP block, in either direction.

This structure is repeated for every LAB in the row (i.e., every LAB has its own set of dedicated R4 interconnects driving left and right). R4 interconnects can drive C4 and C16 interconnects to propagate signals vertically to different rows. They can also drive R24 interconnects to efficiently travel long distances.

The R8 interconnects are identical to the R4 interconnects except that they span 8 blocks instead of 4 and only connect to R8 and C8 interconnects. By design, the R8 interconnect is faster than two R4 interconnects joined together. The R24 interconnect provides the fastest long-distance interconnection. It is similar to the R4 and R8 interconnects, but does not connect directly to the LAB local interconnects. Instead, it is connected to row and column interconnects at every fourth LAB and only communicates to LAB local interconnects through R4 and C4 routes. R24 interconnections connect with all interconnection routes except L8s.

In the column (vertical) dimension, the resources are very similar. They include LUT chain and register chain direct connections and interconnects of lengths 4, 8, and 16 that traverse 4, 8, or 16 blocks up and down, respectively. The LAB local interconnects found in row routing resources are mirrored through LUT chain and register chain interconnects. The LUT chain connects the combinatorial output of one LE to the fast input of the LE directly below it without consuming general routing resources. The register chain connects the register output of one LE to the register input of another LE to implement fast shift registers.

Finally, although this discussion was LAB-centric, all blocks connect to the MultiTrack row and column interconnect using a direct connection similar to the LAB local connection interfaces. These direct connection blocks also support fast direct communication to neighboring LABs.

1.5.2 Xilinx Virtex-II Pro

Launched and shipped right behind the Altera Stratix, the Xilinx Virtex-II Pro FPGA was the flagship product of Xilinx, Inc. for much of 2002 and 2003. A good deal of the information that is presented here is adapted from "Module 2 (Functional Description)" of the October 2005 edition of *Xilinx Virtex-II Pro™ and Virtex-II Pro X™ Platform FPGA Handbook* (available at *http://www.xilinx.com*).

The Virtex-II Pro is an SRAM-based island-style FPGA with several heterogeneous computational elements interconnected through a complex routing matrix. The basic logic tile is the configurable logic block (CLB), consisting of four *slices* and two 3-state buffers. These CLBs are tiled across the device in rows and columns with a segmented, hierarchical interconnect tying all the resources together. Dedicated memory blocks, SelectRAM+, are spread throughout the device. Additional computational resources are provided in dedicated 18×18-bit multiplier blocks.

Logic architecture

The smallest piece of logic from the perspective of the interconnect structure is the CLB. Shown in Figure 1.16, it consists of four equivalent *slices* organized into two columns of two slices each with independent carry chains and a common shift chain. Each slice connects to the general routing fabric through a configurable switch matrix and to each other in the CLB through a fast local interconnect.

Each slice comprises primarily two 4-LUT function generators, two programmable registers for state holding, and fast carry logic. The slice also contains extra multiplexers (MUXFx and MUXF5) to allow a single slice to be configured for wide logic functions of up to eight inputs. A handful of other gates provide extra functionality in the slice, including an XOR gate to complete a 2-bit full adder in a single slice, an AND gate to improve multiplier implementations in the logic fabric, and an OR gate to facilitate implementation of sum-of-products chains.

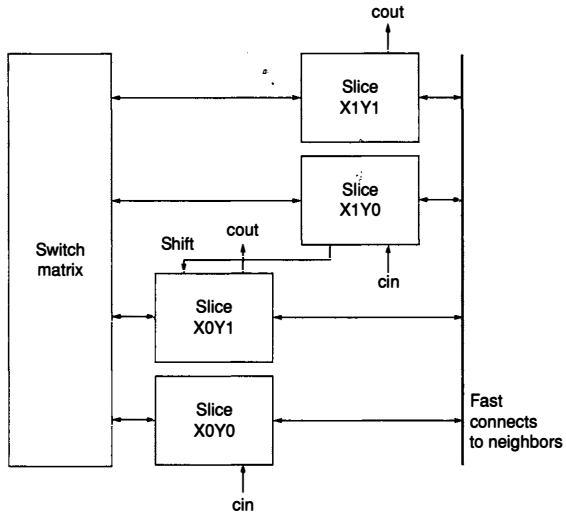


FIGURE 1.16 ■ Xilinx Virtex-II Pro configurable CLB. (*Source:* Adapted from [8], Figure 32, p. 35.)

In the largest Virtex-II Pro device, the XC2VP100, there are 120 rows and 94 columns of CLBs. This translates into 44,096 individual slices and 88,192 4-LUTs—comparable to the largest Stratix device. In addition to these general configurable logic resources, the Virtex-II Pro provides dedicated RAM in the form of block SelectRAM+. Organized into multiple columns throughout the device, each block SelectRAM+ provides 18 Kb of independently clocked, true dual-port synchronous RAM. It supports a variety of configurations, including single- and dual-port access in various aspect ratios. In the largest device there are 444 blocks of block SelectRAM+ organized into 16 columns, yielding a total of 8,183,808 bits of memory.

Complementing the general logic resources are a number of 18×18-bit 2's complement signed multiplier blocks. Like the DSP blocks in the Altera Stratix, these multiplier structures are designed for DSP-type operations, including FIR, IIR, FFT, and others, which often require multiply-accumulate structures. As shown in Figure 1.17, each 18×18 multiplier block is closely associated with an 18Kb block SelectRAM+. The use of the multiplier/block SelectRAM+ memory, with an accumulator implemented in LUTs, allows the implementation of efficient multiply-accumulate structures. Again, in the largest device, just as with block SelectRAM+, there are 16 columns yielding a total of 444 18×18-bit multiplier blocks.

Finally, the Virtex-II Pro has one unique feature that has been carried into newer products and can also be found in competing Altera products. Embedded

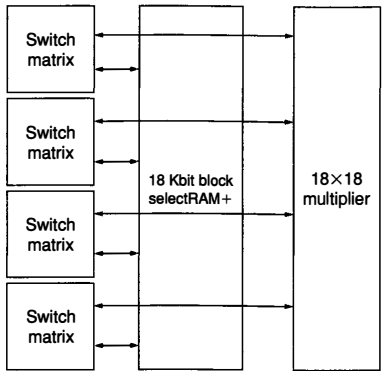


FIGURE 1.17 ■ Virtex-II Pro multiplier/block SelectRAM+ organization. (*Source:* Adapted from [8], Figure 53, p. 48.)

in the silicon of the FPGA, much like the multiplier and block SelectRAM+ structures, are up to four IBM PowerPC 405-D5 CPU cores. These cores can operate up to 300+ MHz and communicate with surrounding CLB fabric, block SelectRAM+, and general interconnect through dedicated interface logic. On-chip memory (OCM) controllers allow the PowerPC core to use block SelectRAM+ as small instruction and data memories if no off-chip memories are available.

The presence of a complete, standard microprocessor that has the ability to interface at a very low level with general FPGA resources allows unique, system-on-a-chip designs to be implemented with only a single FPGA device. For example, the CPU core can execute housekeeping tasks that are neither time-critical nor well suited to implementation in LUTs.

Routing architecture

The Xilinx Virtex-II Pro provides a segmented, hierarchical routing structure that connects to the heterogeneous fabric of elements through a switch matrix block. The routing resources (dubbed Active Interconnect) are physically located in horizontal and vertical routing channels between each switch matrix and look quite different from the Altera Stratix interconnect structures.

The routing resources available between any two adjacent switch matrix rows or columns are shown in Figure 1.18, with the switch matrix block shown in black. These resources include, from top to bottom, the following:

- 24 long lines that span the full height and width of the device.
- 120 hex lines that route to every third or sixth block away in all four directions.

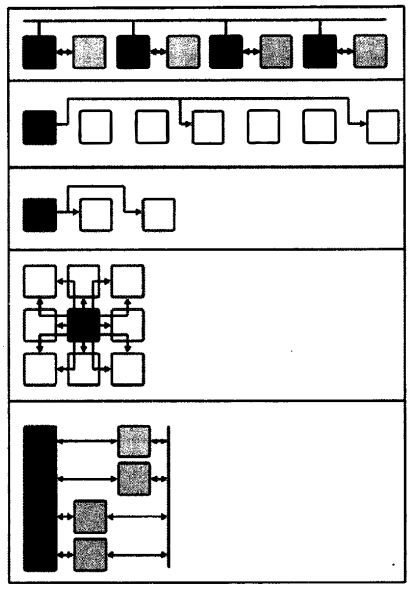


FIGURE 1.18 ■ Xilinx Virtex-II Pro routing resources. (*Source:* Adapted from [7], Figure 54, p. 45.)

- 40 double lines that route to every first or second block away in all four directions.
- 16 direct connect routes that route to all immediate neighbors.
- 8 fast-connect lines in each CLB that connect LUT inputs and outputs.

1.6 SUMMARY

This chapter presented the basic inner workings of FPGAs. We introduced the basic idea of lookup table computation, explained the need for dedicated computational blocks, and described common interconnection strategies. We learned how these devices maintain generality and programmability while providing performance through dedicated hardware blocks. We investigated a number of ways to program and maintain user-defined configuration information. Finally, we tied it all together with brief overviews of two popular commercial architectures, the Altera Stratix and the Xilinx Virtex-II Pro.

Now that we have introduced the basic technology that serves as the foundation of reconfigurable computing, we will begin to build on the FPGA to create

reconfigurable devices and systems. The following chapters will discuss how to efficiently conceptualize computations spatially rather than procedurally, and the algorithms necessary to go from a user-specified design to configuration data. Finally, we'll look into some application domains that have successfully exploited the power of reconfigurable computing.

References

[1] J. Rose, A. E. Gamal, A Sangiovanni-Vincentelli. Architecture of field-programmable gate arrays. *Proceedings of the IEEE* 81(7), July 1993.

[2] P. Chow, et al. The design of an SRAM-based field-programmable gate array—Part 1: Architecture. *IEEE Transactions on VLSI Systems* 7(2), June 1999.

[3] H. Fan, J. Liu, Y. L. Wu, C. C. Cheung. On optimum switch box designs for 2-D FPGAs. *Proceedings of the 38th ACM/SIGDA Design Automation Conference (DAC)*, June 2001.

[4] ———. On optimal hyperuniversal and rearrangeable switch box designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22(12), December 2003.

[5] H. Schmidt, V. Chandra. FPGA switch block layout and evaluation. *IEEE International Symposium on Field-Programmable Gate Arrays*, February 2002.

[6] Xilinx, Inc. *Xilinx XC4000E and XC4000X Series Field-Programmable Gate Arrays, Product Specification* (Version 1.6), May 1999.

[7] Altera Corp. *Altera Stratix™ Device Handbook*, July 2005.

[8] Xilinx, Inc. *Xilinx Virtex-II Pro™ and Virtex-II Pro™ Platform FPGA Handbook*, October 2005.

CHAPTER 2

RECONFIGURABLE COMPUTING ARCHITECTURES

Lesley Shannon
School of Engineering Science
Simon Fraser University

There has been considerable research into possible reconfigurable computing architectures. Alternatives range from systems constructed using standard off-the-shelf field-programmable gate arrays (FPGAs) to systems constructed using custom-designed chips. Standard FPGAs benefit from the economies of scale; however, custom chips promise a higher speed and density for custom-computing tasks. This chapter explores different design choices made for reconfigurable computing architectures and how these choices affect both operation and performance. Questions we will discuss include:

- Should the reconfigurable fabric be instantiated as a separate coprocessor or integrated as a functional unit (see Instruction augmentation subsection of Section 5.2.2)
- What is the appropriate granularity (Chapter 36) for the reconfigurable fabric?

Computing applications generally consist of both control flow and dataflow. General-purpose processors have been designed with a control plane and a data plane to support these two requirements. All reconfigurable computers have a reconfigurable fabric component that is used to implement at least a portion of the dataflow component of an application.

In this discussion, the reconfigurable fabric in its entirety will be referred to as the *reconfigurable processing fabric*, or RPF. The RPF may be statically or dynamically reconfigurable, where *a static* RPF is only configured between application runs and a *dynamic* RPF may be updated during an application's execution.

In general, the reconfigurable fabric is relatively symmetrical and can be broken down into similar tiles or cells that have the same functionality. These blocks will be referred to as *processing elements*, or PEs. Ideally, the RPF is used to implement computationally intensive kernels in an application that will achieve significant performance improvement from the pipelining and parallelism available in the RPF. The kernels are called *virtual instruction configurations*, or VICs, and we will discuss possible RPF architectures for implementing them in the following section.

2.1 RECONFIGURABLE PROCESSING FABRIC ARCHITECTURES

One of the defining characteristics of a reconfigurable computing architecture is the type of reconfigurable fabric used in the RPF. Different systems have quite different granularities. They range from fine-grained fabrics that manipulate data at the bit level similarly to commercial FPGA fabrics, to coarse-grained fabrics that manipulate groups of bits via complex functional units such as ALUs (arithmetic logic units) and multipliers. The remainder of this section will provide examples of these architectures, highlighting their advantages and disadvantages.

2.1.1 Fine-grained

Fine-grained architectures offer the benefit of allowing designers to implement bit manipulation tasks without wasting reconfigurable resources. However, for large and complex calculations, numerous fine-grained PEs are required to implement a basic computation. This results in much slower clock rates than are possible if the calculations could be mapped to fewer, coarse-grained PEs. Fine-grained architectures may also limit the number of VICs that can be concurrently stored in the RPF because of capacity limits.

Garp's nonsymmetrical RPF

The BRASS Research Group designed the Garp reconfigurable processor as an MIPS processor and on-chip cache combined with an RPF [14]. The RPF is composed of an array of PEs, as shown in Figure 2.1. Unlike most RPF architectures, not all of the PEs (drawn as rounded squares in the array) are the same. There is one control PE in each row (illustrated as the dark gray square in the leftmost column) that provides communication between the RPF and external resources. For example, the control block can be used to generate an interrupt for the main processor or to initiate memory transactions. The remaining PEs (illustrated as light gray squares) in the array are used for data processing and modeled after the configurable logic blocks (CLBs) in the Xilinx 4000 series [13]. The number of columns of PEs is fixed at 24, with the middle 16 PEs dedicated to providing memory access for the RPF. The 3 extra PEs on the left and the 4 extra PEs on the right in Figure 2.1 are used for operations such as overflow, error checking, status checking, and wider data sizes.

The number of rows in the RPF is not fixed by the architecture, but is typically at least 32 [13]. A wire network is provided between rows and columns, but the only way to switch wires is through logic blocks, as there are no connections from one wire to another. Each PE operates at the bit level on two bits of data, performing the same operation on both bits based on the assumption that a large fraction of most configurations will be used for multibit operations. By creating identical configurations for both bits, the configuration size and time can be reduced but only at the expense of flexibility [13].

The loading of configurations into an RPF with a fine-grained fabric is extremely costly relative to coarse-grained architectures. For example, each PE

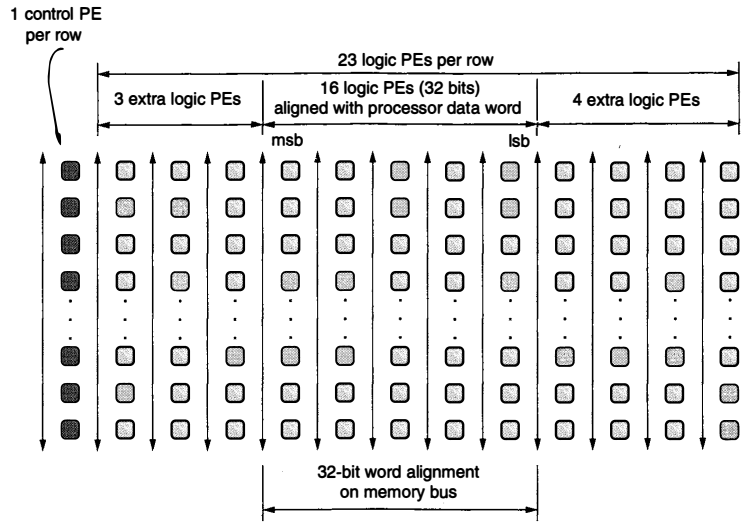


FIGURE 2.1 ■ Garp's RPF architecture. (*Source:* Adapted from [13].)

in Garp's RPF requires 64 configuration bits (8 bytes) to specify the sources of inputs, the PE's function, and any wires to be driven by the PE [13]. So, if there are only 32 rows in the RPF, 6144 bytes are required to load the configuration. While this may not seem significant given that the configuration bitstream of a commercial FPGA is on the order of megabytes (MB), it is considerable relative to a traditional CPU's context switch. For example, if the bit path to external memory from the Garp is assumed to be 128 bits, loading the full configuration takes 384 sequential memory accesses.

Garp's RPF architecture supports partial array configuration and is dynamically reconfigurable during application execution (i.e., a *dynamic* RPF). Garp's RPF architecture allows only one VIC to be stored on the RPF at a time. However, up to four different full RPF VIC configurations can be stored in the on-chip cache [13]. The VICs can then be swapped in and out of the RPF as they are needed for the application.

The loading and execution of configurations on the reconfigurable array is always under the control of a program running on the main (MIPS) processor. When the main processor initiates a computation on the RPF, an iteration counter in the RPF is set to a predetermined value. The configuration executes until the iteration counter reaches zero, at which point the RPF stalls. The MIPS-II instruction set has been extended to provide the necessary support to the RPF [13].

Originally, the user was required to write configurations in a textual language that is similar to an assembler. The user had to explicitly assign data and operations to rows and columns. This source code was fed through a program called the *configurator* to generate a representation for the configuration as a collection of bits in a text file. The rest of the user's source code could then be written in C, where the configuration was referenced using a character array initializer. This required some further assembly language programming to invoke the Garp instructions that interfaced with the reconfigurable array. Since then, considerable compiler work has been done on this architecture, and the user is now able to program the entire application in a high-level language (HLL) [14] (see Chapter 7).

2.1.2 Coarse-grained

For the purpose of this discussion, we describe coarse-grained architectures as those that use a bus interconnect and PEs that perform more than just bitwise operations, such as ALUs and multipliers. Examples include PipeRench and RaPiD (which is discussed later in this chapter).

PipeRench

The PipeRench RPF architecture [6], as shown in Figure 2.2, is an ALU-based system with a specialized reconfiguration strategy (Chapter 4). It is used as a coprocessor to a host microprocessor for most applications, although applications such as PGP and JPEG can be run on PipeRench in their entirety [8]. The architecture was designed in response to concerns that standard FPGAs do not provide reasonable forward compatibility, compilation time, or sufficient hardware to implement large kernels in a scalable and portable manner [6].

The PipeRench RPF uses pipelined configuration, first described by Goldstein et al. [6], where the reconfigurable fabric is divided into physical pipeline stages that can be reconfigured individually. Thus, the resulting RPF architecture is both partially and dynamically reconfigurable. PipeRench's compiler is able to compile the static design into a set of "virtual" stages such that each virtual stage can be mapped to any physical pipeline stage in the RPF. The complete set of virtual stages can then be mapped onto the actual number of physical stages available in the pipeline. Figure 2.3 illustrates how the virtual pipeline stages of an application can be mapped onto a PipeRench architecture with three physical pipeline stages.

A pipeline stage can be loaded during each cycle, but all cyclic dependencies must fit within a single stage. This limits the types of computations the array can support, because many computations contain cycles with multiple operations. Furthermore, since configuration of a pipeline stage can occur concurrent to execution of another pipeline stage, there is no performance degradation due to reconfiguration.

A row of PEs is used to create a physical stage of the pipeline, also called a physical stripe, as shown in Figure 2.2. The configuration word, or VIC, used to configure a physical stripe is also known as a virtual stripe. Before a physical

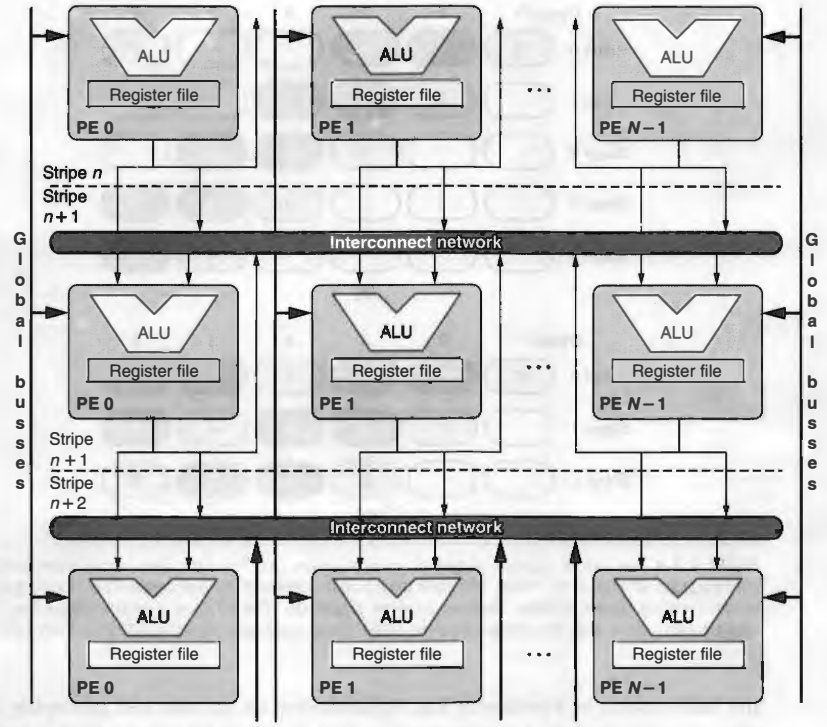


FIGURE 2.2 ■ PipeRench architecture: PEs and interconnect. (*Source:* Adapted from [6].)

stripe is configured with a new virtual stripe, the state of the present virtual stripe, if any, must be stored outside the fabric so it can be restored when the virtual stripe is returned to the fabric. The physical stripes are all identical so that any virtual stripe can be placed onto any physical stripe in the pipeline. The interconnect between adjacent stripes is a full crossbar, which enables the output of any PE in one stage to be used as the input of any PE in the adjacent stage [6].

The PEs for PipeRench are composed of an ALU and a pass register file. The pass register file is required as there can be no unregistered data transmitted over the interconnect network between stripes, creating pipelined interstripe connections. One register in the pass register file is specifically dedicated to intrastripe feedback. An 8-bit PE granularity was chosen to optimize the performance of a suite of kernels [6].

It has been suggested that reconfigurable fabric is well suited to stream-based functions (see Chapter 5, Section 5.1.2) and custom instructions [6]. Although

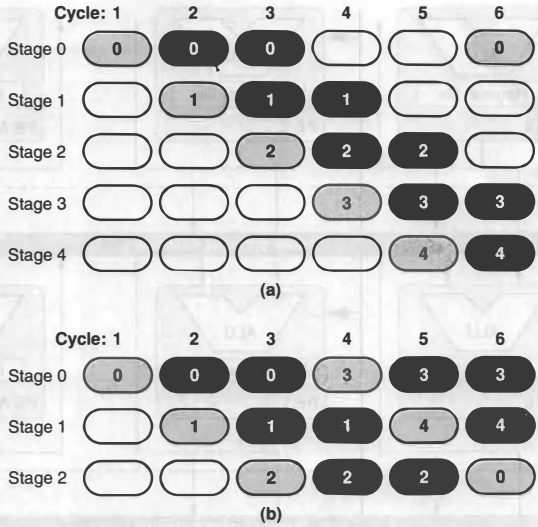


FIGURE 2.3 ■ The virtual pipeline stages of an application (a). The light gray blocks represent the configuration of a pipeline stage; the dark gray blocks represent its execution. The mapping of virtual pipeline stages to three physical pipeline stages (b). The physical pipeline stages are labeled each cycle with the virtual pipeline stage being executed. (*Source:* Adapted from [6].)

the first version of PipeRench was implemented as an attached processor, the next was designed as a coprocessor so that it would be more tightly coupled with the host processor [6]. However, the developers of PipeRench argue against making the RPF a functional unit on the host processor. They state that this could "restrict the applicability of the reconfigurable unit by disallowing state to be stored in the fabric and in some cases by disallowing direct access to memory, essentially eliminating their usefulness for stream-based processing" [6].

PipeRench uses a set of CAD tools to synthesize a stripe based on the parameters N, B, and P, where N is the number of PEs in the stripe, B is the width in bits of each PE, and P is the number of registers in a PE's pass register file. By adjusting these parameters, PipeRench's creators were able to choose a set of values that provides the best performance according to a set of benchmarks [6]. Their CAD tools are able to achieve an acceptable placement of the stripes on the architecture, but fail to achieve a reasonable interconnect routing, which has to be optimized by hand.

The user also has to describe the kernels to be executed on the PipeRench architecture using the *Dataflow Intermediate Language* (DIL), a single-assignment C-like language created for the architecture. DIL is intended for use by programmers and as an intermediate language for any high-level language compiler

that targets PipeRench architectures [6]. Obviously, applications have to be recompiled, and probably even redesigned, to run on PipeRench.

2.2 RPF INTEGRATION INTO TRADITIONAL COMPUTING SYSTEMS

Whereas the RPF in a reconfigurable computing device dictates the programmable logic resources, a full reconfigurable computing system typically also has a microprocessor, memory, and possibly other structures. One defining characteristic of reconfigurable computing chips is the integration, or lack of integration, of the RPF with a host CPU.

As shown in Figure 2.4, there are multiple ways to integrate an RPF into a computing system's memory hierarchy. The different memory components of the system are drawn as shaded rectangles, where the darker shading indicates a tighter coupling of the memory component to the processor. The types of RPF integration for these computing systems are illustrated as rounded

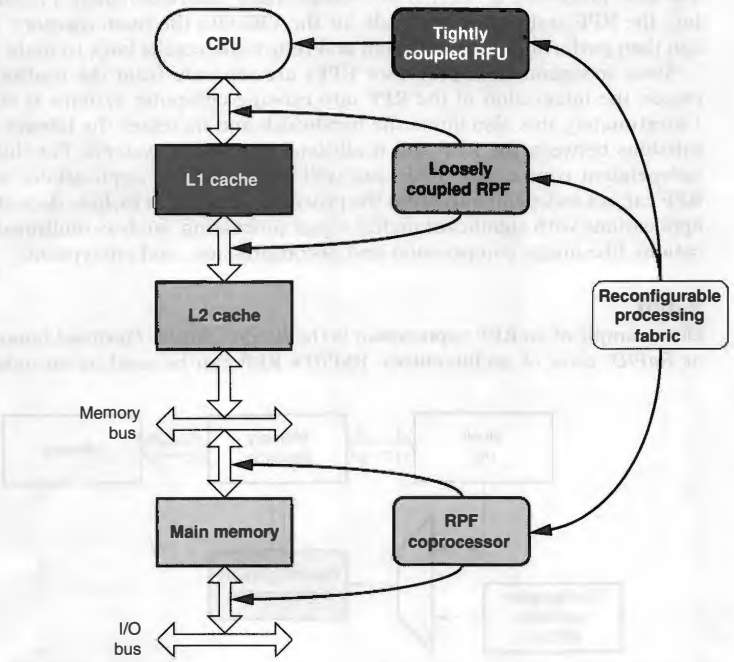


FIGURE 2.4 ■ Possible locations for the RPF in the memory hierarchy. (*Source:* Adapted from [6].)

rectangles, where the darker shading indicates a tighter coupling of the RPF to the processor. Some systems have the RPF as a separate processor [2–7]; however, most applications require a microprocessor somewhere to handle complex control. In fact, some separate reconfigurable computing platforms are actually defined to include a host processor that interfaces with the RPF [1]. Unfortunately, when the RPF is integrated into the computing system as an independent coprocessor, the limited bandwidth between CPU and reconfigurable logic can be a significant performance bottleneck.

Other systems include an RPF as an extra functional unit coupled with a more traditional processor core on one chip [8–24]. How tightly the RPF is coupled with the processor's control plane varies.

2.2.1 Independent Reconfigurable Coprocessor Architectures

Figure 2.5 illustrates a reconfigurable computing architecture with an independent RPF [1–7]. In these systems, the RPF has no direct data transfer links to the processor. Instead, all data communication takes place through main memory. The host processor, or a separate configuration controller, loads a configuration into the RPF and places operands for the VIC into the main memory. The RPF can then perform the computation and return the results back to main memory.

Since independent coprocessor RPFs are separate from the traditional processor, the integration of the RPF into existing computer systems is simplified. Unfortunately, this also limits the bandwidth and increases the latency of transmissions between the RPF and traditional processing systems. For this reason, independent coprocessor RPFs are well suited only to applications where the RPF can act independently from the processor. Examples include data-streaming applications with significant digital signal processing, such as multimedia applications like image compression and decompression, and encryption.

RaPiD

One example of an RPF coprocessor is the *Reconfigurable Pipelined Datapaths* [4], or *RaPiD*, class of architectures. RaPiD's RPF can be used as an independent

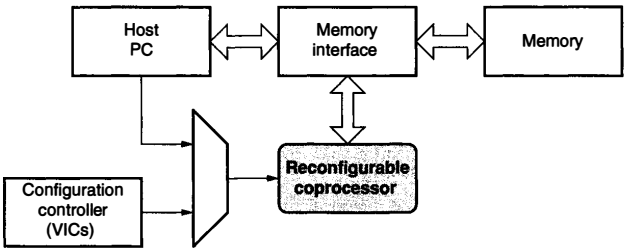


FIGURE 2.5 ■ A reconfigurable computing system with an independent reconfigurable coprocessor.

coprocessor or integrated with a traditional computing system as shown in Figure 2.5. RaPiD is designed for applications that have very repetitive pipelined computations that are typically represented as nested loops [5]. The underlying architecture is comparable to a super-scalar processor with numerous PEs and instruction generation decoupled from external memory but with no cache, no centralized register file, and no crossbar interconnect, as shown in Figure 2.6.

Memory access is controlled by the *stream generator*, which uses first-in-first-out (FIFOs), or *streams* (Chapter 5, Sections 5.1.2 and 5.2.1), to obtain and transfer data from external memory via the memory interface, as shown in Figure 2.7. Each stream has an associated address generator, and the individual address patterns are generated statically at compile time [5]. The actual reads and writes

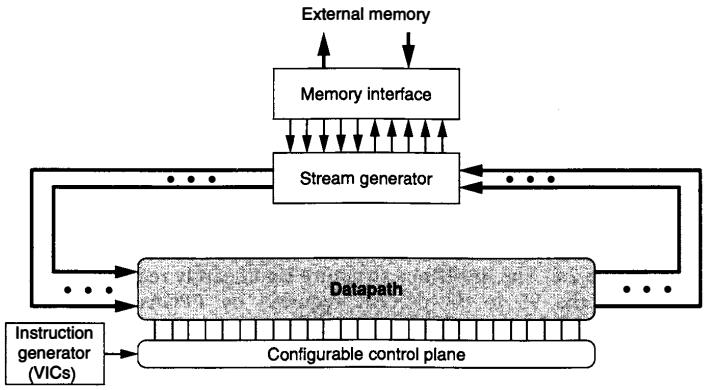


FIGURE 2.6 ■ A block diagram of the RaPiD architecture (*Source:* Adapted from [5].)

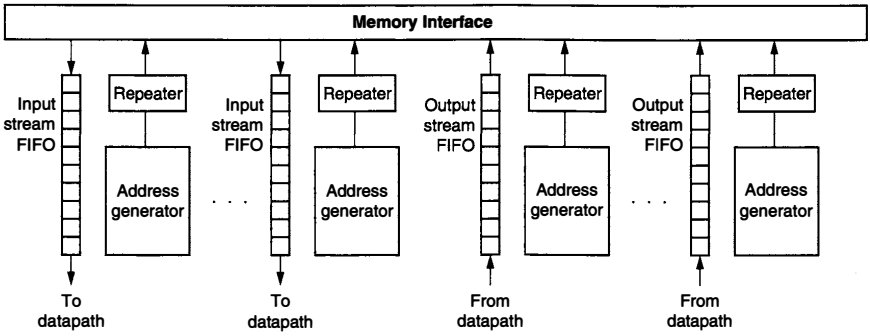


FIGURE 2.7 ■ RaPiD's stream generator. (*Source:* Adapted from [5].)

from the FIFOs are triggered by instruction bits at runtime. If the datapath's required input data is not available (i.e., the input FIFO is empty) or if the output data cannot be stored (i.e., the output FIFO is full), then the datapath will stall. Fast access to memory is therefore important to limit the number of stalls that occur. Using a fast static RAM (SRAM), combined with techniques, such as interleaving and out-of-order memory accesses, reduces the probability of having to stall the datapath [5].

The actual architecture of RaPiD's datapath is determined at fabrication time and is dictated by the class of applications that will be using the RaPiD RPF. This is done by varying the PE structure and the data width, and by choosing between fixed-point or floating-point data for numerical operations. The ability to change the PE's structure is fundamental to RaPiD architectures, with the complexity of the PE ranging from a simple general-purpose register to a multi-output booth-encoded multiplier with a configurable shifter [5].

The RaPiD datapath consists of numerous PEs, as shown in Figure 2.8. The creators of RaPiD chose to benchmark an architecture with a rather complex PE consisting of ALUs, RAMs, general-purpose registers, and a multiplier to provide reasonable performance [5]. The coarse-grained architecture was chosen because it theoretically allows simpler programming and better density [5]. Furthermore, the datapath can be dynamically reconfigured (i.e., a dynamic RPF) during the application's execution.

Instead of using a crossbar interconnect, the PEs are connected by a more area-efficient linear-segmented bus structure and bus connectors, as shown in Figure 2.8. The linear bus structure significantly reduces the control overhead—from the 95 to 98 percent required by FPGAs to 67 percent [5]. Since

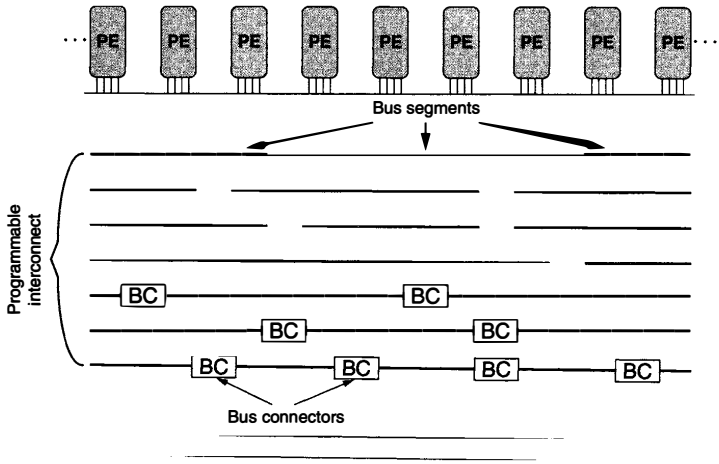


FIGURE 2.8 ■ An overview of RaPiD's datapath. (*Source:* Adapted from [5].)

the processor performance was benchmarked for a rather complex PE, the datapath was composed of only 16 PEs [5].

Each operation performed in the datapath is determined by a set of control bits, and the outputs are a data word plus status bits. These status bits enable data-dependent control. There are both *hard* control bits and *soft* control bits. As the hard control bits are for static configuration and are field programmable via SRAM bits, they are time consuming to set. They are normally initialized at the beginning of an application and include the tristate drivers and the programmable routing bus connectors, which can also be programmed to include pipelined delays for the datapath. The soft control bits can be dynamically configured because they are generated efficiently and affect multiplexers and ALU operations. Approximately 25 percent of the control bits are soft [5].

The instruction generator generates soft control bits in the form of VICs for the configurable control plane, as shown in Figure 2.9. The RaPiD system is built around the assumption that there is regularity in the computations. In other words, most of its processing time is spent within nested loops, as opposed to initialization, boundary processing, or completion [5], so the soft control bits are generated by a small programmable controller as a short instruction word (i.e., a VIC).

The programmable controller is optimized to execute nested loop structures. For each nested loop, the user's original code is statically compiled to remove all conditionals on loop variables and expanded to generate static instructions for loops [5]. The innermost loop can then often be packed into a single VIC with a count indicating how many times the VIC should be issued. One VIC can also be used to control more than one operation in more than one pipeline stage [5]. Figure 2.10(a) shows a snippet of code that includes conditional statements (if and for). This same functionality is shown in terms of static instructions in Figure 2.10(b).

As there are often parallel loop nests in applications, the instruction generator has multiple programmable controllers running in parallel (see Figure 2.9) [5]. Although this causes synchronization concerns, the appropriate status bits exist to provide the necessary handshaking. The VICs from each controller are

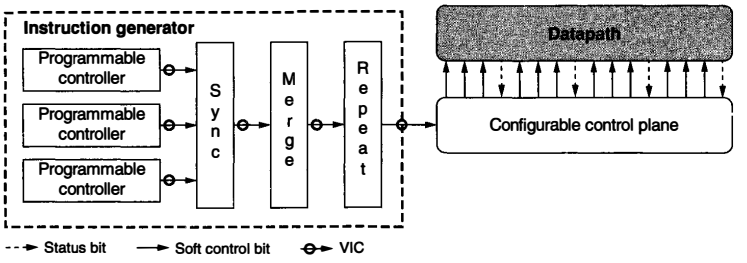


FIGURE 2.9 ■ RaPiD's instruction generator. (*Source:* Adapted from [5].)

```
for (i=0; i<10; i++)
{
      for(j=0; j<16; j++)
      {
            if(j==0)
                  load data;
            else if(j < 8)
                  x = x + y;
            else
                  z = y * z;
      }
}
```

(a)

```
Execute 10 times
{

      Execute once:          // j==0 case
            load data;
      Execute six times:     // 0<j<8 case
            x = x + y;
      Execute eight times:    // 7<j<16 case
            z = y * z;

}
```

(b)

FIGURE 2.10 ■ Original code (a) and pseudo-code (b) for static instruction implementation of the original code.

synchronized to ensure proper coordination between the parallel loops and then merged to generate the configurable control plane for the entire datapath [5].

There are obvious benefits to RaPiD, but it is not easily programmed: The programmer must use a specialized language and compiler designed specifically for RaPiD. This allows the designer to specify the application in such a way as to obtain better hardware utilization [5]. However, this class of architecture is not well suited to highly irregular computations with complex addressing patterns, little reuse of data, or an absence of fine-grained parallelism, which do not map well to RaPiD's datapath [5].

It is interesting to note that while RaPiD was implemented as a stand-alone processor, its creators suggest that it would be better to combine RaPiD with an RISC engine on the same chip so that it would have a larger application space [5]. The RISC processor could control the overall computation flow, and RaPiD could speed up the compute-intensive kernels found in the application. The developers also suggest that better performance could be achieved if RaPiD were a special functional unit as opposed to a coprocessor, because it would be more closely bound to the general-purpose processor [5]. These are the types of architecture we will be discussing in the following section.

2.2.2 Processor + RPF Architectures

As opposed to the independent coprocessor model, other systems more tightly couple the RPF with the host processor on the same chip; in some cases, the RPF is loosely coupled with the processor as an independent functional unit. Such architectures typically allow direct access to the RPF from the processor as well as independent access to memory, as do the Garp architecture [13] and the Chameleon system [20] (to be discussed in the following section). Alternatively, we can couple the RPF more tightly with the processor. For example, in architectures, such as Chimaera [18] (to be discussed later in this chapter), the

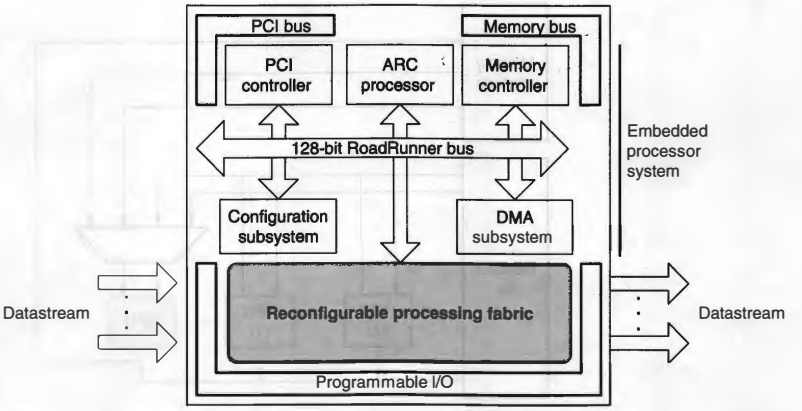


FIGURE 2.11 ■ Chameleon's RCP architecture. (*Source:* Adapted from a figure obtained off of Chameleon System's home page, which is no longer available.)

RPF is incorporated as a reconfigurable functional unit (RFU) (see Instruction augmentation subsection in Section 5.2.2) within the processor itself.

Loosely coupled RPF and processor architecture

The commercial Reconfigurable Communications Processor (RCP) was created by Chameleon Systems Inc. [20]. It combined an embedded processor subsystem with an RPF via a proprietary shared bus architecture, known as the RoadRunner bus (Figure 2.11). The RPF had direct access to the processor as well as direct memory access (DMA). The reconfigurable fabric also had a programmable I/O interface so that users could process off-chip I/O independent of the rest of the embedded on-chip processing system. This provided more flexibility for the RPF than in typical reconfigurable computing architectures, where the RPF generally had access only to the processor and memory.

The Chameleon architecture was able to provide improved price/performance relative to the highest-performing DSPs of its time, but its RCP consumed more power because of the RPF. After 2002, there was little mention of Chameleon or its RCP. Conceptually, the product was an interesting idea, but it failed to corner a product niche during the electronics market downturn.

Tightly coupled RPF and processor

Figure 2.12 illustrates a traditional processor's datapath architecture with the RPF integrated as an RFU. Such systems tightly couple the RFU to the central processing unit's (CPU) datapath similarly to the technology of traditional CPU functional units (FUs), such as the ALU, the multiplier, and the FPU. In some cases, these architectures only provide RFU access to input data from the register file in the same way as the traditional CPU FUs (Chimaera [18], PRISC

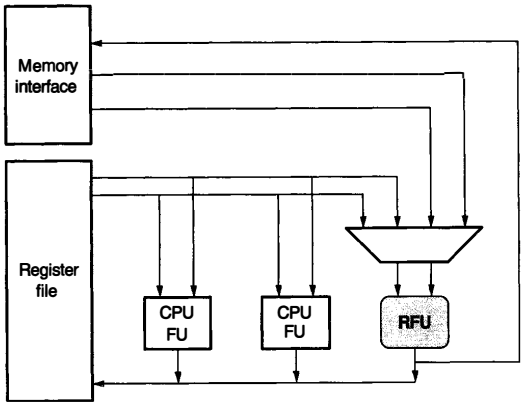


FIGURE 2.12 ■ The datapath of the processo. + RFU architecture.

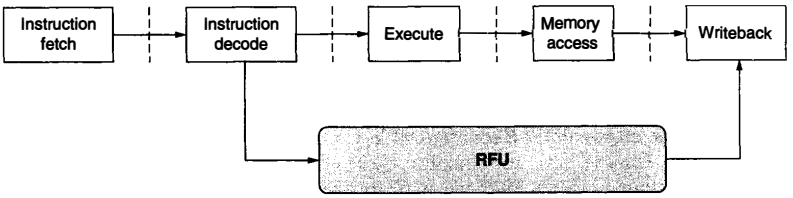


FIGURE 2.13 ■ An example of a pipeline of a processor with an RFU. (*Source:* Adapted from [16].)

[11], etc.). Other architectures allow the RFU to access data stored in the local cache/memory directly (e.g., OneChip [16]). Many of them can have multiple VICs instantiated in the RFU at once, enabling designers to accelerate multiple software instructions at the same time.

For reconfigurable computing architectures in which the RFU is tightly coupled with the processing core, the processor pipeline must be updated as shown in Figure 2.13. VICs in the RFU typically run during the *execute* stage (and possibly the *memory* stage) of the pipeline. Some of these processors are capable of running VICs in parallel with instructions that use more traditional processor resources, such as the ALU or FPU, and even support out-of-order execution (OneChip [16], Chimaera [18]).

Chimaera

The Chimaera architecture [18], shown in Figure 2.14, was developed at Northwestern University. Its developers created a C compiler that could create

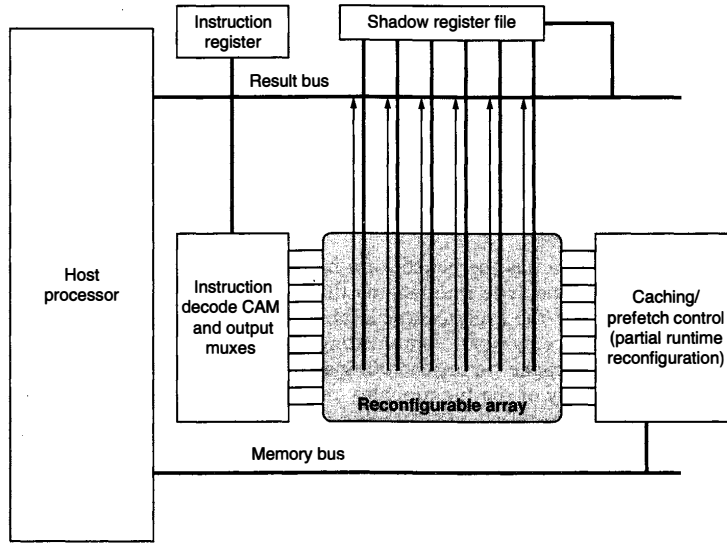


FIGURE 2.14 ■ Overview of the Chimaera architecture. (*Source:* Adapted from [18].)

specialized instructions for their RFU, known as RFUOPs (VICs for the purpose of our discussion) [19]. These custom instructions are created on a per application basis and have direct access to the processor's register file. Furthermore, commonly used VICs can be cached for easy reloading so that the processor does not have to stall while the RFU is configured [19].

The RFU is structured as a reconfigurable array (RA) of PEs, where any VIC occupies an integer number of rows. Influenced by the Triptych FPGA [18], the Altera Flex 8000 series, and the PRISC architecture [11], the array structure is FPGA-like to support computationally intensive kernels. Each PE in a row operates on 1 bit, with each row containing the same number of PEs as the size of the processor's memory word. The RFU can be partially configured so that multiple VICs can be cached in it at any given time. When an instruction is to be written to the RFU and there are no empty rows, the VIC that is overwritten is chosen such that configurations of the RFU will be minimized [19].

Another benefit of the Chimaera architecture is that it allows for speculative execution of VICs. Any VIC that is loaded in the RFU speculatively executes each cycle. If one of them is actually executed, the resulting value is stored at the writeback stage; otherwise, it is ignored and discarded. The RFU also supports multi-input operations, so that any VIC occupying one row will execute in a single clock cycle and with the appropriate data dependencies. Assuming that

data dependencies are not an issue, multi-cycle operations can execute without pipelining stalls [19].

When a VIC is detected at the decode stage of the pipeline, a check is made of the RFU to determine if it is already loaded. If it is not loaded, a check is made of the VIC cache. If the VIC instruction is not in either of these locations, it must be loaded from memory to reconfigure the necessary rows of the RFU. In that case the microprocessor will stall. This is time consuming because, although the precise configuration timing requirements are not specified, the objective is to minimize the number of configurations of the RFU performed from memory [19].

Chimaera has the benefit of a high-level design language for the user. It also has the same style interface as that of a normal stand-alone processor, which means that the architecture is able to provide extra functionality to improve performance, without complicating the design process. The idea is to treat the RFU as a cache for instructions as opposed to logic and then to assume that the majority of the functionality required for the algorithm will be supplied by the microprocessor [18]. In this way, the RFU can be used to accelerate the program's computationally intensive kernels. Integrating the RPF as an RFU within the processor has increased the bandwidth for communication between the two [18]. However, because the RFU cannot access memory directly, it is overly dependent on the host processor to fetch and store operands.

2.3 SUMMARY AND FUTURE WORK

In this chapter, we discussed key characteristics of reconfigurable computing architectures and their tradeoffs; specifically: (1) how the RPF should be coupled into the system, and (2) what the nature of the RPF should be. Fine-grained fabrics allow users to perform bitwise operations without wasting reconfigurable resources, whereas basic multibit computations can be mapped to fewer coarse-grained modules and run at a faster clock rate.

The coupling of the RPF with a traditional processor affects both its ability to do independent computation and the rate at which data can be transferred from the processor itself. Independent reconfigurable coprocessors are easily added to a traditional processing system and can operate independently from the processor. However, this loose coupling increases the latency and decreases the communication bandwidth between the processor and the RPF. In contrast, tightly coupling the RPF to the processor facilitates communication and data transfers, but limits the RFP's independence. In tightly coupled architectures, the RPF is often part of the processor's pipeline, potentially stalling execution until the VIC is completed. Loosely coupled RPFs try to offer the best of both worlds: sufficient independence from the main processor to prevent pipeline stalls combined with reasonable bandwidth for inter-processor/RPF communications.

One important challenge in developing reconfigurable computing architectures is to create CAD tools and programming environments that enable designers to use HLLS. This would allow designers to abstract the low-level hardware

of the RPF and to simplify programming the architecture, while still achieving speedup over a traditional processor. Another significant challenge is how to evaluate reconfigurable computing architectures. There is no equivalent to the Spec Benchmark [25] set for such evaluation. Furthermore, as these architectures may have different programming models or limited compiler support, designers are not easily able to run the same benchmark on multiple architectures for a standard comparison.

That Chameleon, and many other reconfigurable computing startup companies in similar market niches, was forced to close its doors during the electronic market downturn in the early 2000s illustrates an interesting aspect of reconfigurable computing as a whole. Even though, theoretically, special-purpose reconfigurable computing chips are a compelling technology, to date they have failed to achieve commercial success and there have been numerous failures. Many popular arguments have been used to justify this failure—they are too power-hungry; an effective high-level programming environment has not been developed; no one has identified a "killer" application to justify the design cost of using them—but no definitive answer exists. As it becomes increasingly difficult to improve the performance of traditional processor architectures, the possibility that reconfigurable computing architectures may yet find their place in the world of commercial success increases.

Despite the lack of significant market success to date, reconfigurable computing is still an area of significant ongoing research and commercial interest. For example, Rapport Inc.'s Kilocore design is a commercial derivative of the PipeRench architecture. As of 2007, Rapport was offering 256 PE components organized as 16 stripes, each composed of 16 8-bit PEs, and it has plans to expand its offerings to components containing thousands of PEs.

References

[1] J. M. Arnold. The Splash 2 software environment. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1993.

[2] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, S. Ghosh. PRISM-II compiler and architecture. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1993.

[3] M. J. Wirthlin, B. L. Hutchings. A dynamic instruction set computer. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1995.

[4] C. Ebeling, D. C. Cronquist, P. Franklin. RaPiD: Reconfigurable Pipelined Datapath. *Proceedings of the Sixth International Workshop on Field-Programmable Logic and Applications*, Springer-Verlag, September 1996.

[5] D. Cronquist, P. Franklin, C. Fisher, M. Figueroa, C. Ebeling. Architecture design of reconfigurable pipelined datapaths. *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, March 1999.

[6] S. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, R. Laufer. PipeRench: A coprocessor for streaming multimedia acceleration. *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.

[7] H. Schmit. Incremental reconfiguration for pipelined applications. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1997.
[8] Y. Chou, P. Pillai, H. Schmit, J. Shen. PipeRench implementation of the instruction path coprocessor. *Proceedings of the 33rd International Symposium on Microarchitecture*, December 2000.
[9] M. J. Wirthlin, B. L. Hutchings, K. L. Gilson. The nano processor: A low resource reconfigurable processor. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1994.
[10] M. Budiu. Application-specific hardware: Computing without CPUs. *Fourth CMU Symposium on Computer Systems*, October 2001.
[11] R. Razdan, M. Smith. A high-performance microarchitecture with hardware-programmable functional units. *Proceedings of the 27th Annual IEEE/ACM International Symposium on Microarchitecture*, November 1994.
[12] B. Kastrup, A. Bink, J. Hoogerbrugge. ConCISe: A compiler-driven CPLD-based instruction set accelerator. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1999.
[13] J. Hauser, J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1997.
[14] T. J. Callahan, J. R. Hauser, J. Wawrzynek. The Garp architecture and C compiler. *Computer*, April 2000.
[15] R. D. Wittig, P. Chow. OneChip: An FPGA processor with reconfigurable logic. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, March 1996.
[16] J. E. Carrillo, E. P. Chow. The effect of reconfigurable units in superscalar processors. *Proceedings of the Ninth ACM International Symposium on Field-Programmable Gate Arrays*, February 2001.
[17] C. R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. A. Arnold, M. Gokhale. The NAPA adaptive processing architecture. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.
[18] S. Hauck, T. W. Fry, M. Hosier, J. P. Kao. The Chimaera reconfigurable functional unit. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1997.
[19] Z. A. Ye, A. Moshovos, S. Hauck, P. Banerjee. CHIMAERA: A high-performance architecture with a tightly coupled reconfigurable functional unit. *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.
[20] D. Wilson. Chameleon takes on FPGAs, ASICs. *Electronic Business Asia, EDN Online Magazine* (*http://www.edn.com/article/CA50551.html?partner=enews*), October 2000.
[21] P. Graham, B. Nelson. Reconfigurable processors for high-performance, embedded digital signal processing. *Proceedings of the Ninth International Workshop on Field-Programmable Logic and Applications*, August 1999.
[22] B. Salefski, L. Caglar. Reconfigurable computing in wireless. *Proceedings of the Design Automation Conference*, June 2001.
[23] T. Bijlsma, P. T. Wolkotte, G. J. M. Smit. An optimal architecture for a DDC. *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)—12th Reconfigurable Architecture Workshop (RAW 2006)*, April 2006.
[24] A. A. Chien, J. H. Byun. Safe and protected execution for the Morph/AMRM reconfigurable processor. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1999.
[25] Standard Performance Evaluation Corp. Spec Benchmarks (*http://www.spec.org*).

CHAPTER 3

RECONFIGURABLE COMPUTING SYSTEMS

Steven A. Guccione
Cmpware, Inc.

Like most technologies, reconfigurable computing systems are built on a variety of existing technologies and techniques. It is always difficult to pinpoint the exact moment a new area of technology comes into existence or even to pinpoint which is the first system in a new class of machines. Popular scientific history often gives simple accounts of individuals and projects that represent a turning point for a particular technology, but in reality the story is usually more complicated. A number of individuals may arrive at similar conclusions, at very nearly the same time, and the details of their research are nearly always different. It is in the investigation of these details that a better understanding of the technology, and its development, can be reached.

While it is satisfying to say that Thomas Edison invented the lightbulb in 1879, the real story is much more complex and much more interesting. Such is the case with reconfigurable computing hardware systems, as it is with most technologies. In the short time that these systems have been in existence, a relatively large number of them, developed by many highly trained and talented individuals from diverse fields, have evolved very quickly. In approximately a decade the number of implemented reconfigurable systems went from a small handful to hundreds.

The large number of exotic high-performance systems designed and built over a very short time makes this area particularly difficult to document, but there is also a problem specific to them. Much of the work was done inside various government agencies, particularly in the United States, and was never published. In these cases, all that can be relied on is currently available records and publications.

3.1 EARLY SYSTEMS

The generally agreed on criterion for a reconfigurable computing system is that it be built from reconfigurable computing devices such as field-programmable gate arrays (FPGAs) or FPGA-like devices. In general, these devices must be reprogrammable and permit hardwarelike levels of performance, if not hardwarelike structures. Moreover, they should permit orders of magnitude speedup over traditional microprocessors for similar clock speeds, silicon

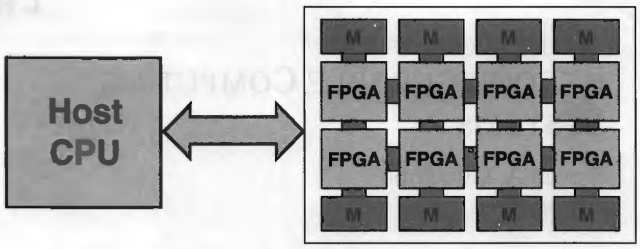


FIGURE 3.1 ■ The traditional processor/coprocessor arrangement for reconfigurable computing hardware.

area, and technology. Most significantly, however, the system must be reprogrammable and able to perform with a variety of applications. Systems that use a fixed hardware design, even if they use reconfigurable computing elements, are viewed more as using this design in place of traditional hardware for cost savings or convenience. It is in the ability to use reconfigurable devices for more general-purpose computing that makes them "reconfigurable."

Reconfigurable systems are likewise distinguished from other cellular multiprocessor systems. Array processors, in particular Single Instruction Multiple Data Processors (SIMDs), are considered architecturally distinct from reconfigurable machines, in spite of many similarities. This distinction arises primarily from the programming techniques. Array processors tend to have either a shared or dedicated instruction sequencer and take standard instruction-style programming code. Reconfigurable machines tend to be programmed spatially, with different physical regions of the device being configured at different times. This necessarily means that they will be slower to reprogram than cellular multiprocessor systems but should be more flexible and achieve higher performance for a given silicon area.

One of the earliest acknowledged reconfigurable computing machines, although it is frequently referenced under "distributed computing," is the Fixed-Plus-Variable (F+V) computer developed by Estrin and his colleagues at the University of California at Los Angeles in the mid-1960s [17–20]. The F+V consisted of a standard processor unit that controlled many other "variable" units. It had several limitations, including the need to manually change wiring as part of the reconfiguration process, but it did offer relatively mature software tools for its time. Generally because of its use of reconfigurable computing concepts, the F+V system is acknowledged to be the forerunner of modern reconfigurable computing architectures.

After the F+V, there was a gap of nearly two decades before more modern reconfigurable computing systems began to be explored. The rise of the modern era began in the mid-1980s, when commercially available FPGA devices from companies such as Xilinx and Altera as well as several smaller companies became widely available.

These devices were generally based around small lookup tables (LUTs) and a programmable interconnection network. The LUTs were typically 8- or 16-bit memories configured to implement arbitrary logic functions, taking their inputs from and sending their outputs to a programmable interconnection network. While this network could not provide arbitrary interconnections, software tools were usually able to produce operational digital circuits for a wide range of popular designs.

Even by 1990, however, the largest FPGA devices supported designs on the order of 10K logic gates. This is a very small number and barely suitable for a parallel multiplier circuit. Even worse, the FPGAs were in competition with modern microprocessors, which were doubling in performance every 18 months and providing a simpler programming model, more mature tools, and a larger base of experienced users. For these reasons, the early work in reconfigurable systems necessarily concentrated on two areas, often simultaneously:

- The systems would have to use relatively large numbers of FPGAs, sometimes hundreds, to achieve sufficient computing power to be of use when compared to microprocessor-based systems.
- They would attack problems that were naturally ill suited to modern microprocessors, including bit-oriented algorithms that did not map efficiently to word-oriented microprocessors and highly structured and repetitive algorithms such as graphics that mapped well to the hardwarelike structures of reconfigurable systems.

The 1990s also marked the beginning of an explosive growth in circuit density following Moore's Law, with a doubling in FPGA density approximately every 18 months. As the density increased, the typical application went from simple interface or "glue" logic circuits to more complex designs, eventually supporting large custom coprocessors, typically for digital signal processing (DSP) or other data-intensive applications. With large, high-quality, commercially available FPGA devices now in use, and with the ongoing rapid increase in density, FPGA-based reconfigurable computing machines quickly became widely available.

3.2 PAM, VCC, AND SPLASH

In the late 1980s, PAM, VCC, and Splash—three significant general-purpose systems using multiple FPGAs—were designed and built. They were similar in that they used multiple FPGAs, communicated to a host computer across a standard system bus, and were aimed squarely at reconfigurable computing.

3.2.1 PAM

The Programmable Active Memories (PAM) project at Digital Equipment Corporation (DEC) initially used four Xilinx XC3000-series FPGAs as shown in Figure 3.2 [8]. The original Perle-0 board contained 25 Xilinx XC3090 devices

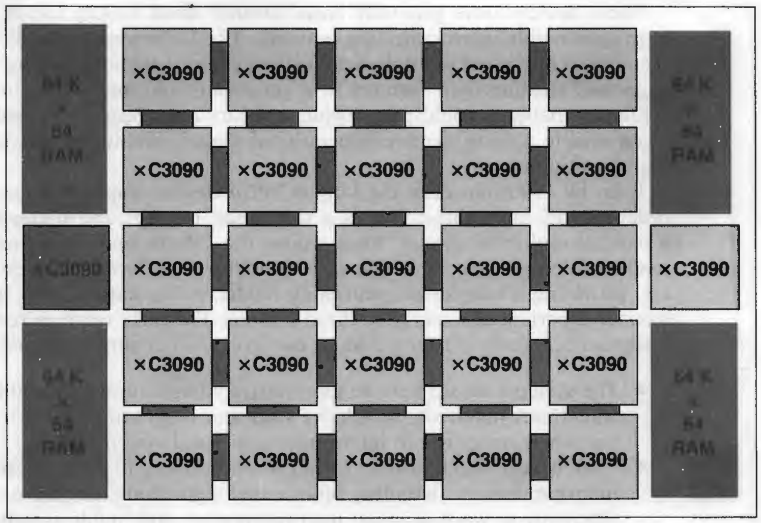


FIGURE 3.2 ■ Digital Equipment Corporation's PAM Perle-0.

in a 5×5 array, attached to which were four independent banks of fast static RAM (SRAM), arranged as 64K × 64 bits, which were controlled by an additional two XC3090 FPGA devices. This wide and fast memory provided the FPGA array with high bandwidth. The Perle-0 was quickly upgraded to the more recent XC4000 series. As the size of the available XC4000-series devices grew, the PAM family used a smaller array of FPGA devices, eventually settling on 2×2.

Based at the DEC research lab, the PAM project ran for over a decade and continued in spite of the acquisition of DEC by Compaq and then the later acquisition of Compaq by Hewlett-Packard. PAM, in its various versions, plugged into the standard PCI bus in a PC or workstation and was marked by a relatively large number of interesting applications as well as some groundbreaking work in software tools. It was made available commercially and became a popular research platform.

3.2.2 Virtual Computer

The Virtual Computer from the Virtual Computer Corporation (VCC) was perhaps the first commercially available reconfigurable computing platform. Its original version was an array of Xilinx XC4010 devices and I-Cube programmable interconnect devices in a checkerboard pattern, with the I-Cube devices essentially serving as a crossbar switch as shown in Figure 3.3 [11]. The topology of the interconnection for these large FPGA arrays was an important issue at this time: With a logic density of approximately 10K gates and input/output (I/O) pins on the order of 200, a major concern was communication across FPGAs. The I-Cube

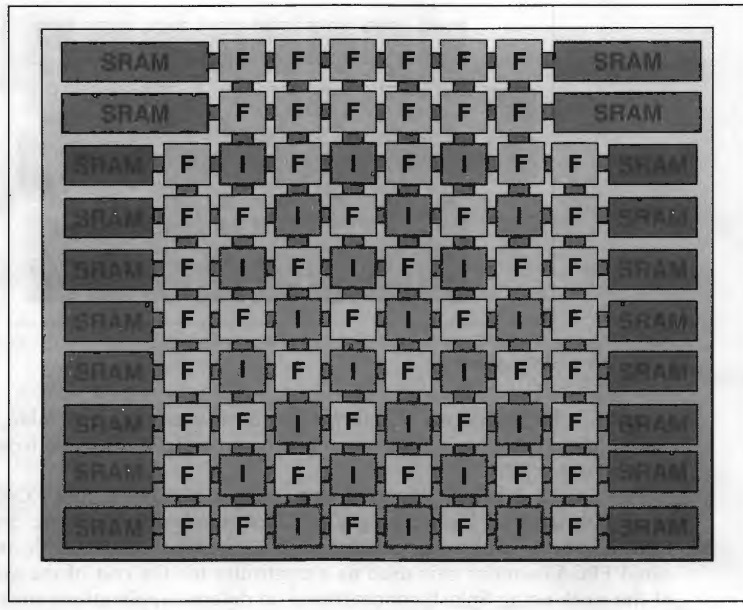


FIGURE 3.3 ■ VCC's Virtual Computer.

devices were perceived as providing more flexibility, although each switch had to be programmed, which increased the design complexity.

The first Virtual Computer used an 8×8 array of alternating FPGA and I-Cube devices. The exception was on the left and right sides of the array, which exclusively used FPGAs, which consumed 40 Xilinx XC4010 FPGAs and 24 I-Cubes. Along the left and right sides were 16 banks of independent 16×8K dual-ported SRAM, and attached to the top row were 4 more banks of standard single-ported $256\text{K} \times 32$ bits SRAM controlled by an additional 12 Xilinx XC4010 FPGAs. While this system was large and relatively expensive, and had limited software support, VCC went on to offer several families of reconfigurable systems over the next decade and a half.

3.2.3 Splash

The Splash system, from the Supercomputer Research Center (SRC) at the Institute for Defense Analysis, was perhaps the largest and most heavily used of these early systems [22,23,27]. Splash was a linear array consisting of XC3000-series Xilinx devices interfacing to a host system via a PCI bus. Multiple boards could be hosted in a single system, and multiple systems could be connected together. Although the Splash system was primarily built and used by the Department of

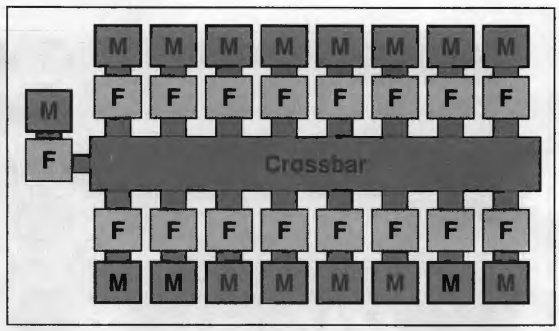


FIGURE 3.4 ■ SRC's Splash 2.

Defense, a large amount of information on it was made available. A Splash 2 system quickly followed and was made commercially available from Annapolis Microsystems [30].

The Splash 2 board consisted of two rows of eight Xilinx XC4010 devices, each with a small local memory attached as shown in Figure 3.4. These 16 FPGA/memory pairs were connected to a crossbar switch, with another dedicated FPGA/memory pair used as a controller for the rest of the system. Much of the work using Splash concentrated on defense applications such as cryptography and pattern matching, but the associated tools effort was also notable, particularly some of the earliest high-level language (HLL) to hardware description language (HDL) translation software targeting reconfigurable machines [4]. Specifically, the data parallel C compiler and its debug tools and libraries provided reconfigurable systems with a new level of software support.

PAM, VCC, and Splash represent the early large-scale reconfigurable computing systems that emerged in the late 1980s. They each had a relatively long lifetime and were upgraded with new FPGAs as denser versions became available. Also of interest is the origin of each system. One was primarily a military effort (Splash), another emerged from a corporate research lab (PAM), and the third was from a small commercial company (Virtual Computer). It was this sort of widespread appeal that was to characterize the rapid expansion of reconfigurable computing systems during the 1990s.

3.3 SMALL-SCALE RECONFIGURABLE SYSTEMS

PAM, VCC, and Splash were large and relatively expensive machines. Because a single high-density FPGA device could cost several thousand dollars, their circuit boards were perhaps some of the most expensive built at that time. Around 1990, a cost of approximately $1 per reconfigurable logic gate in a reconfigurable

system was not unusual. Of course, this cost dropped rapidly as FPGAs of higher densities and lower prices became available.

Because of their cost, the use of FPGAs was somewhat limited, and none of these systems achieved widespread success as consumer products. While work on them was ongoing, smaller-scale FPGAs were beginning to appear that would have a major impact on the direction of the field, especially because their rapidly increasing density meant that the large multichip systems of yesterday would very soon fit within a single device.

3.3.1 PRISM

One of the smaller-scale experiments with reconfigurable computing was PRISM, developed at Brown University [5]. This was an unusual project in that it used a single small FPGA as a coprocessor in a larger distributed system. This distributed processor/coprocessor arrangement was unique for its time and would reappear many years later in more mainstream reconfigurable supercomputers. It permitted small but often complex calculations to be offloaded from the central processing unit (CPU) to the reconfigurable coprocessor. The circuits implemented in the coprocessor may not have been large, but the tighter coupling to the processor gave this architecture an advantage in places where larger and more expensive arrays would not have been appropriate.

Also of note are PRISM's software development tools. Its compiler technology used was advanced for its era and was one of the earlier experiments in high-level language programming of reconfigurable systems. In particular, it addressed a more fine-grained form of coprocessing where the host CPU and the reconfigurable coprocessor shared the workload. Larger systems tended to have vastly more powerful reconfigurable units and often used the host only for simple control, input/output, and display. The workload was seldom shared with the host CPU in any meaningful way in these larger systems.

3.3.2 CAL and XC6200

Perhaps the most interesting project of this era came from Algotronix, a small Scottish company with connections to the University of Edinburgh [28], which created its own FPGA exclusively targeted at reconfigurable computing [1]. The Configurable Array Logic (CAL) featured very simple logic cells compared to other commercial FPGAs. What was unique about CAL was that each cell could be individually addressed and reconfigured dynamically—something that no other FPGA device at the time could manage. CAL also featured a fairly standard bus interface that permitted it to be easily used with a microprocessor in a coprocessor arrangement. Algotronix also offered some fairly traditional graphical tools to program CAL, as well as a small board containing multiple CAL devices.

While CAL was unique and influential, it was not until the acquisition of Algotronix by Xilinx in the early 1990s that its ideas would become more widespread. Xilinx began development of a second-generation CAL device it called the XC6200 [12]. Many of its features were the same as those of the earlier CAL

devices, but the backing of Xilinx gave the XC6200 a high level of acceptance, at least in the research community.

Perhaps the most groundbreaking aspect of CAL, and later the XC6200 family, was largely nontechnical. Because these devices supported fine-grained and dynamic reconfiguration, they required that the configuration bitstream be openly documented. Thus, all of the internal details of the logic circuitry and the configuration process were fully documented and made publicly available. Unlike other programmable hardware, the internal programming codes for most FPGAs have never been published, largely for historical and practical reasons.

3.3.3 Cloning

Early in the history of FPGAs, there was some concern that lower-cost "cloned" FPGA devices could be made by third parties. For instance, a company could produce a device that was functionally identical to a Xilinx XC4000 and sell it to Xilinx customers. This was common practice for older and smaller silicon devices and was sometimes encouraged by manufacturers. However, the large investment FPGA vendors had in software tools and silicon intellectual property made them resistant to releasing any more information than necessary about their silicon architectures. Also, as long as the high-level design tools were available for a reasonable price and worked well, most users did not have any particular need to examine an FPGA device's internal workings.

Today it is unlikely that a device as complex as an FPGA could be "cloned." Even if the technical challenges could be overcome, legal barriers to using such intellectual property probably could not be successfully challenged.

Another concern of some customers was that knowledge of the internal workings of FPGA devices could permit their designs to be compromised. While most people familiar with the issues tended to dismiss the idea of reverse-engineering, especially as FPGAs have increased in size, it was still a concern to some customers.

For these reasons, FPGA bitstreams have traditionally been, and still remain, a tightly held trade secret. The XC6200 broke ranks by publicizing its configuration data and permitting a new level of experimentation with tools and applications that could make use of these powerful new modes of operation. Commercial success for the XC6200 would be elusive in the fiercely competitive and rapidly changing FPGA market of the 1990s, but it remained a favorite of researchers even long after its cancellation by Xilinx.

3.4 CIRCUIT EMULATION

One of the early large-scale uses of reconfigurable logic was for circuit emulation. Because FPGAs can, in theory, implement any arbitrary digital logic circuit, some people realized that they could be used as a form of simulation accelerator. At the time, digital circuit simulation had become a bottleneck in the design process. As integrated circuit designs became larger and more complex, the

time necessary to simulate them also grew. Without accurate simulation, design errors inevitably crept into the final design, often requiring another expensive and time-consuming redesign cycle. In spite of the high cost of FPGA devices, the ability to quickly and accurately evaluate the function of large and complex digital circuits became very valuable. Also, for chip designs that included programmable processors (or that were the processors themselves), an FPGA-based prototype provided a development platform for testing the software that would eventually run on the production device.

Interestingly, the larger and more complex the circuit, the more difficult and time consuming simulation became and the more valuable FPGA-based emulation would be to designers. For this reason, some of the largest and most expensive FPGA-based machines have traditionally been digital circuit emulators. Some purists may point out that such machines were highly application specific and did not necessarily constitute reconfigurable computing. While these machines did often simulate only a single design in their lifetimes, they were usually reconfigured as much as several times per day to perform different functions. Also, in some cases users would go on to realize that the emulation platforms could be employed for more general-purpose computing.

Emulation using reconfigurable logic quickly became popular, with very large-scale systems becoming commercially available in rapid succession. PiE and QuickTurn in the United States announced their machines, as did the smaller InCA in Europe. The machines were very similar, all attempting to put as many high-density FPGAs as possible into a single system. Because they were highly scalable, and their densities and prices were changing rapidly, it is difficult to gauge what a typical large FPGA emulation system would be. However, a system on the order of 1 million programmable logic gates built from devices with approximately a 10K-gate capacity would be representative of a large, but early FPGA emulation. While large and expensive, these systems were very valuable to integrated circuit designers, who knew the high cost of designs with bugs. One place in particular where they had a large impact was in the design of microprocessors.

3.4.1 AMD/Intel

Because microprocessors were very complex and had strict deadlines to meet, emulation became very important at places such as Advanced Micro Devices (AMD) and Intel. And because the new microprocessor parts often had to be compatible with older models, emulation was a very good way to guarantee that systems would be compatible across generations. One event in the 1990s would help further drive emulator popularity. AMD and Intel began a decades-long competition to produce the latest high-performance device compatible with a x86 instruction set for the desktop PC market.

Initially, AMD was in the "follower" position and was attempting to create functionally identical versions of Intel devices. This was no small challenge, and with new products being released almost yearly, the value to AMD of getting a functionally correct Intel work-alike device as soon as possible was very high.

Emulators played a very large role in verifying the functional correctness of the AMD designs against the Intel device. While the emulated designs would run at perhaps a few hundred kilohertz as compared to the tens of megahertz of the final silicon devices, being able to run test vectors at this rate, and even eventually booting entire operating systems, was crucial in proving the compatibility of these microprocessor designs.

Emulation is still widely used in digital design, but the increasing size and decreasing cost of FPGA devices has led to a smaller market for very large emulation machines such as the ones offered by QuickTurn and PiE. In fact, QuickTurn and PiE merged in 1993 after a short legal battle. The merged company was acquired in 1998 by Cadence, a CAD software vendor.

3.4.2 Virtual Wires

Although emulation was largely a commercial endeavor, one research project in this area warrants special mention—the Virtual Wires Project (see Chapter 30, Logic Emulation on Multi-FPGA Systems) at M.I.T., which produced an emulator that helped overcome one of the most serious limitations of emulators of the time [6]. Whereas the logic density of FPGAs grew rapidly, chip-to-chip interconnect soon became the limiting factor in large, multi-FPGA designs such as emulation. In fact, many emulated designs used only a fraction of the logic in the FPGAs while consuming all of the input/output resources. Then along came Virtual Wires with a pin multiplexing scheme to share I/O pins on FPGA devices transparently, permitting their higher utilization. This technology would be licensed to another logic emulation company, Ikos, which would eventually be bought by another of the large CAD software vendors, Mentor Graphics.

Emulation had perhaps two major impacts on reconfigurable computing. First, it was an early large-scale user of reconfigurable logic that was commercially successful. This helped drive similar work in the field. Perhaps just as important, many of the researchers involved in the emulation work saw the value of more general-purpose computing using reconfigurable logic and would go on to lead advancements in other areas of reconfigurable systems.

3.5 ACCELERATING TECHNOLOGY

After the success of digital circuit emulators and the research results of the early systems, reconfigurable computing was poised to expand. Three factors helped drive this expansion. First, the ever-increasing density of FPGA devices was making larger and larger amounts of reconfigurable logic available at an increasingly lower price. In just a few short years, the million-gate systems that took several large boards could be built with a single device. This in itself led to widespread experimentation with reconfigurable computing as dozens of research projects at universities and research labs across the globe sprang up.

The second factor is one that has become more obvious in retrospect. By the mid-1990s the decades-long increase in microprocessor computing power was

beginning to ebb. Late in the decade, it was clear that manufacturing technology constraints, power consumption issues, and architectural limitations such as memory performance were bringing an end to the long era of microprocessor dominance. In the past, new solutions to high-performance computing had had to contend with the yearly appearance of a new microprocessor with double the performance of the previous generation and a consumer-friendly price. This made it difficult for custom high-performance systems to be competitive. With the end of the steep growth in microprocessor performance in sight, however, other solutions to high performance were beginning to look more attractive. Reconfigurable computing technology happened to be emerging just at this critical juncture and would be considered by many as a top contender for the future of high-performance computing.

The third factor was the Department of Defense's new funding program, named Adaptive Computing Systems (ACS), which invested more than $100 million in reconfigurable computing research during the mid- to late 1990s. It is always difficult to judge the effect of such a program, but it is clear that ACS not only led to an increased level of research in this field but also provided a useful forum for researchers, both academically and commercially. While the program funded exclusively U.S. researchers, it also appears to have spurred reconfigurable computing research in other places, particularly the United Kingdom and Japan [31, 32].

The era of expansion in reconfigurable computing technology was marked by a rapid growth in the number of systems being constructed. An accurate count of projects in this area is difficult, but certainly dozens and perhaps hundreds of reconfigurable systems were constructed at this time [25]. However, the increased density of FPGA devices led to a shift away from large, expensive systems like those of the first generation and toward smaller systems, often containing a single FPGA device on a standard board to be plugged into a personal computer or workstation.

The new systems tended to be primarily for research and were more often than not hobbled by two problems. First, the tools to program a reconfigurable computing platform were not standardized and often amounted to two completely decoupled design flows. Hardware design tools provided by the FPGA vendor were used to construct a circuit in the FPGA coprocessor, while standard software development tools were used to program the host PC or workstation. This hardware/software codesign style was inefficient and inflexible, and required highly skilled engineers. For those reasons, although there were a few notable software and tools projects at this time, they were more the exception than the rule. None achieved widespread popularity.

3.5.1 Teramac

Among the projects to come out of this era, Teramac [3, 14], a product of the Hewlett-Packard research laboratories, bears special mention, for three reasons. First, it went against the trend by creating a large multi-FPGA machine. Second, it straddled different markets by being aimed at both circuit emulation and

reconfigurable computing. Lastly, it was constructed of custom-integrated circuits instead of commercially available FPGA devices.

Teramac was originally designed to perform emulation for a large microprocessor design that was being developed jointly by Hewlett-Packard and Intel. It was to be the first 64-bit Intel processor and at the time went by the name "Merced." The joint Intel/HP project was announced in 1994 and was expected to produce its first silicon device by 1999.

All of this was taking place just as large circuit emulators from vendors such as QuickTurn were emerging as the new tools for large microprocessor development. The HP/Intel venture decided to also produce its own emulator, which would not use commercial FPGAs but rather an HP custom-designed reconfigurable logic device [2]. This was not as unusual an idea as it may seem. Intel and HP certainly had the resources to produce such a machine, and the current FPGA-based offerings were far from perfect.

The three biggest problems associated with emulators at this time were cost, low circuit density, and tools. In fact, the tools problem was perhaps the most severe of the three. Large designs needed massive computing resources on their own to be converted into configuration bitstreams for the many FPGA devices. If we assume that the emulator hardware consisted of hundreds of FPGA devices, each taking several hours of time on a standard personal computer or workstation to produce a configuration bitstream, it is clear that a large computational resource was required just to produce the data used by the emulator.

This part of design and test was often the bottleneck, and there appeared to be little that could be done to accelerate the process. Additionally, commercial FPGA devices were aimed at a more general-purpose logic design market and were not explicitly aimed at emulation. A special-purpose device more tailored to the needs of circuit emulation could provide the higher density and performance required by emulation users.

Teramac was announced in 1995 and had some unique features. First, it successfully overcame many of the limitations of the commercial FPGA devices of that era. Its custom FPGA (called Plasma) focused on fast compilation times via very flexible crossbar-based interconnects. This was in contrast to commercial FPGA's focus on logic density and performance, and it meant that the placement and routing of a design for a single Plasma device took seconds, not minutes to hours. Perhaps more interesting, Plasma made good use of defect tolerance. Boards and devices that would otherwise have been thrown away could be used in the Teramac; an analysis phase would test the system to log defects and permit the faulty portions of the system to be bypassed. While regular array architectures such as FPGAs lend themselves naturally to such defect and fault tolerance, it had not traditionally been used in commercial reconfigurable logic devices.

In addition to its emulation duties, Teramac was used for applications such as image processing, bioinformatics, search, and CAD, making it a true reconfigurable computing platform. However, while Teramac was successful, the chip it was built to emulate, the IA-64 family, was somewhat less so. The IA-64 devices were late to market, but they did eventually ship and found their way

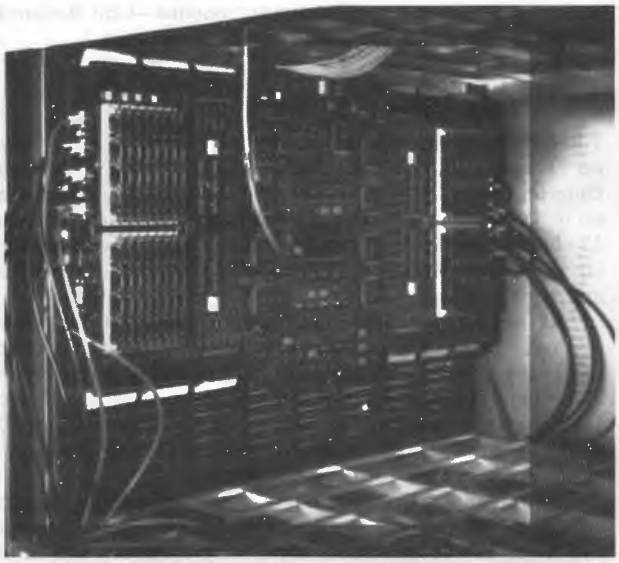


FIGURE 3.5 ■ A Hewlett-Packard Laboratories Teramac board.

into commercial products—just not enough to justify the massive investment by HP and Intel, which would not jointly produce other architectures. Thus, Teramac became an early casualty of the HP/Intel microprocessor design partnership. Figure 3.5 shows a picture of one of the boards from a Teramac system.

3.6 RECONFIGURABLE SUPERCOMPUTING

While the number of small reconfigurable coprocessing boards would continue to proliferate as commercial FPGA devices became denser and cheaper, other new hardware architectures were produced to address the needs of large-scale supercomputer users. Unlike the earlier generation of boards and systems that sought to put as much reconfigurable logic as possible into a single unified system, these machines took a different approach. In general, they were traditional multiprocessor systems, but each processing node in them consisted of a very powerful commercial desktop microprocessor combined with a large commercial FPGA device. Another factor that made these systems unique is that they were all offered by mainstream commercial vendors. By 2005 the three largest

makers of traditional supercomputer systems—Cray Research, SRC, and Silicon Graphics—were all producing systems of this type.

3.6.1 Cray, SRC, and Silicon Graphics

The first reconfigurable supercomputing machine from Cray, the XD1, is based on a chassis of 12 processing nodes, with each node consisting of an AMD Opteron processor. Up to 6 reconfigurable computing processing nodes, based on the Xilinx Virtex-4 devices, can also be configured in each chassis, and up to 12 chassis can be combined in a single cabinet, with multiple cabinets making larger systems. Hundreds of processing nodes can be easily configured with this approach.

SRC, a company with historic connections to Cray, takes a more aggressive approach to reconfigurable computing [34]. Both of their multiprocessor systems feature traditional processor and reconfigurable processing units that share a common buslike structure and may be mixed in various configurations [21]. Like the Cray system, the SRC machines also use large Xilinx Virtex-series FPGAs and x86-family desktop processors. SRC also offers smaller personal workstation systems for development.

Finally, Silicon Graphics offers its Reconfigurable Application-Specific Processor (RASP) family of systems [36], which also use high-density Xilinx Virtex FPGAs as its reconfigurable computing elements, but in dual-device configurations on a "blade"-style module. These are very small boards that can be plugged into large racks, often with the system still operating. They interface to the more traditional Silicon Graphics workstation and multiprocessor systems, which also use high-performance desktop microprocessors but are based on the MIPS architecture.

The Cray, SRC, and Silicon Graphics machines point to a clear direction for large-scale reconfigurable computing systems. They combine a more distributed array of FPGA elements with an emphasis on floating-point arithmetic. As FPGA densities continue to increase, the ability to perform large floating-point calculations, even multiple floating-point calculations, in a single device becomes significant. Also, as the performance of commodity microprocessors remains plateaued, it is likely that acceleration techniques such as those used in these reconfigurable machines will continue to be used.

3.6.2 The CMX-2X

A discussion of distributed, floating-point FPGA-based supercomputing would not be complete without a mention of the CM-2X [13]. This machine predates the current crop of reconfigurable supercomputers by over a decade and consists of a Connection Machine 2 from Thinking Machines supplemented with FPGA coprocessors instead of the standard floating-point devices typically used. The CM-2X was a defense-related project, and little information is available on it. However, along with the PRISM system, it is clearly the forerunner of this family of distributed multiprocessor reconfigurable supercomputers.

3.7 NON-FPGA RESEARCH

Although the vast majority of reconfigurable computing systems were based on commercially available FPGA devices, there are some notable exceptions. A small number of projects designed and built custom-reconfigurable silicon devices as the basis of their designs [7, 15, 16, 24, 26, 33, 35]. The general trend was to replace the smaller-grained LUTs in the FPGA architecture with coarser-grained structures more amenable to computing. Typically this meant arithmetic logic units (ALUs) that mapped more closely to traditional programming languages.

Such a coarser-grained approach raises the issue of categorizing non-FPGA devices. Large numbers of ALU-like structures quickly begin to resemble multiprocessors or very long instruction word (VLIW) machines more than they do FPGAs. The way routing is performed may further differentiate non-FPGA from FPGA devices. In general, non-FPGAs are computation, not circuit, oriented. They can easily produce the larger and more complex circuits used by typical arithmetic-based computations, but may not be able to efficiently implement arbitrary digital logic functions.

These systems may have broken new and interesting ground, but the problem with them may ultimately be a practical one. Because commercial FPGAs are very popular, they tend to use the latest silicon processes and are very efficiently designed. The software support for such devices is also decidedly nontrivial. To produce a custom reconfigurable computing device that can compete with both the dense, efficient circuitry and the large body of available software tools of modern FPGAs is a daunting prospect. Given these barriers, no serious contenders to commercial FPGAs as the basis for reconfigurable computing machines have arisen. While the ideas behind these novel architectures are sound and the advantages tangible, it has proved difficult to offer them as a viable alternative to FPGA-based reconfigurable systems.

3.8 OTHER SYSTEM ISSUES

In spite of nearly two decades of intensive research and commercial activity, and the potential to provide orders of magnitude performance, reconfigurable logic-based computing systems have not yet begun to displace conventional systems in any significant way. There are perhaps many factors in this lack of acceptance, but technical details at the hardware level certainly appear to be one of the most serious.

One unavoidable architectural problem involves the necessary use of reconfigurable logic in a processor/coprocessor arrangement, which ties an inherently serial host system to the high-performance and highly parallel reconfigurable processing unit. This connection is necessarily made across a system bus of some sort, which is guaranteed to serialize access to the coprocessor. Thus, the reconfigurable coprocessor can only be "fed" data at a relatively low and fixed

rate. Such a drawback resembles the "von Neumann bottleneck" in conventional uniprocessor systems, where access to memory over a similar bus restricts performance. In the case of reconfigurable systems, the bus interface is the same but the processor is connected to the reconfigurable unit instead of to a memory unit.

By a similar analogy, Amdahl's Law states that an algorithm's parallel performance is eventually dominated by its serial portions. If, for instance, an algorithm is 90 percent parallelizable, the limit on speedup is 10. This implies that even if the parallel portion of the algorithm can be executed in zero time, the serial portion will still take the same fixed amount of time to execute. Similarly, no matter how much work can be offloaded to the reconfigurable coprocessors, the portions that cannot will tend to dominate the computation time.

In this sense, the same problems that limit the ability to parallelize algorithms also limit the ability to use reconfigurable computing. While there are other issues that limit acceptance of reconfigurable systems, including the lack of mature software development tools and competition from other, more conventional architectures, the basic inability to exploit the parallelism in general-purpose reconfigurable computing will always be a serious concern.

The conventional desktop or server approaches to reconfigurable systems have their difficulties, but reconfigurable computing may still find an agreeable environment in embedded systems, which tend to have streaming data inputs and outputs and may not be at the mercy of the bandwidth of existing system buses. In addition, there may be other attractive features of reconfigurable logic in such embedded systems, including lower overall power consumption and the ability to dynamically adapt to external conditions.

3.9 THE FUTURE OF RECONFIGURABLE SYSTEMS

There appear to be some clear trends in the relatively brief, but active, history of reconfigurable computing. Commercial FPGA devices have continued to be dominant in such systems, but FPGA architectures are also evolving, beginning to incorporate coarser-grained resources. Block memory units and multiplier units have become standard, and even multiple microprocessor cores have found their way onto FPGA devices. Morover, this trend has been mirrored in the coarser-grained research efforts in more recent reconfigurable logic devices. Clearly there is a trend toward coarser-grained elements, as well as a heterogeneous variety of elements.

Perhaps in a related way, large-scale high-performance computing, or supercomputing, has clearly embraced reconfigurable logic. Reconfigurable computing appears to be the path to the higher levels of performance desired by these architectures, particularly as traditional microprocessor architectures have reached a performance plateau. Still, while the manufacturers of supercomputing equipment have clearly embraced reconfigurable computing, it remains to be seen if end users will do so as well.

References

[1] Algotronix, Ltd. *CAL1024 Datasheet*, 1990.

[2] R. Amerson, R. Carter, W. Culbertson, P. Kuekes, G. Snider. Plasma: An FPGA for million gate systems. *Proceedings of the ACM/SIGDA Fourth International Symposium on Field-Programmable Gate Arrays*, February 1996.

[3] R. Amerson, R. Carter, W. Culbertson, P. Kuekes, G. Snider. Teramac-configurable custom computing. *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1995.

[4] J. M. Arnold. The Splash 2 software environment. *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1993.

[5] P. M. Athanas, H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer* 26(3), March 1993.

[6] J. A. Babb, R. Tessier, A. Agarwal. Virtual wires: Overcoming pin limitations in FPGA-based logic emulators. *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1993.

[7] V. Baumgarten, F. May, A. Nuckel, M. Vorbach, M. Weinhardt. PACT XPP—A self-reconfigurable data processing architecture. *First International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, June 25–28, 2001.

[8] P. Bertin, D. Roncin, J. Vuillemin. Introduction to programmable active memories. *Technical Report 3*, DEC Paris Research Laboratory, 1989.

[9] D. H. Brown Assoc. Cray XD1 brings high-bandwidth supercomputing to the mid-market (*http://www.cray.com/downloads/dhbrown_crayxd1_oct2004.pdf*), October 2004.

[10] D. A. Buell, K. L. Pocek, eds. *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, IEEE Computer Society Press, 1993.

[11] S. Casselman. Virtual computing and the virtual computer. *IEEE Workshop on FPGAs for Custom Computing Machines*, April 1993.

[12] S. Churcher, T. Kean, B. Wilkie. XC6200 FASTMAP™ processor interface. *Proceedings of the Fifth International Workshop on Field-Programmable Logic and Applications, FPL 1995*, August/September 1995.

[13] S. A. Cuccaro, C. F. Reese. The CM-2X: A hybrid CM-2/Xilinx prototype. *IEEE Workshop of FPGAs for Custom Computing*, April 1993.

[14] W. B. Culbertson, R. Amerson, R. J. Carter, P. J. Kuekes, G. Snider. Teramac configurable custom computer. *Field-Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, Proceedings of International Society of Optical Engineering*, October 1995.

[15] C. Ebeling, D. C. Cronquist, P. Franklin. RaPiD—Reconfigurable pipelined datapath. *Field-Programmable Logic: Smart Applications, New Paradigms and Compilers*, R. W. Hartenstein, M. Glesner, eds., Springer-Verlag, September 1996.

[16] C. Ebeling, D. C. Cronquist, P. Franklin, J. Secosky, S. G. Berg. Mapping applications to the rapid configurable architecture. *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1997.

[17] G. Estrin. Organization of computer systems—The fixed plus variable structure computer. *Proceedings of the Western Joint Computer Conference*, May 1960.

[18] G. Estrin, B. Bussell, R. Turn, J. Bibb. Parallel processing in a restructurable computer system. *IEEE Transactions on Electronic Computers* 12(5), December 1963.

[19] G. Estrin, R. Turn. Automatic assignment of computations in a variable structure computer system. *IEEE Transactions on Electronic Computers* 12(5), December 1963.

[20] G. Estrin, C. R. Viswanathan. Organization of a "fixed-plus-variable" structure computer for eigenvalues and eigenvectors of real symmetric matrices. *Journal of the ACM* 9(1), January 1962.

[21] O. D. Fidanci, D. Poznanovic, K. Gaj, T. El-Ghazawi, N. Alexandritis. Performance overhead in a hybrid reconfigurable computer. *Reconfigurable Architecture Workshop*, April 2003.

[22] M. Gokhale, W. Holmes, A. Kosper, D. Kunze, D. Lopresti, S. Lucas, R. Minnich, P. Olsen. SPLASH: A reconfigurable linear logic array. *International Conference on Parallel Processing*, 1990.

[23] M. Gokhale, A. Kosper, S. Lucas, R. Minnich. The logic description generator. *Proceedings of the International Conference on Application Specific Array Processing*, 1990.

[24] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R. Taylor. PipeRench: A reconfigurable architecture and compiler. *IEEE Computer* 33(4), April 2000.

[25] S. A. Guccione. List of FPGA-based computing machines (*http://www.io.com/~guccione/HW_list.html*), 1994.

[26] R. W. Hartenstein, M. Herz, T. Hoffmann, U. Nageldinger. Using the KressArray for configurable computing. *Proceedings of the International Society of Optical Engineering Conference on Configurable Computing: Technology and Applications*, November 1998.

[27] M. W. Holmes, A. Kosper, S. Lucas, R. Minnich, D. Sweely. Building and using a highly parallel programmable logic array. *IEEE Computer* 24(1), January 1991.

[28] T. A. Kean. *Configurable Logic: A Dynamically Programmable Cellular Architecture and Its VLSI Implementation*, Ph.D. thesis, University of Edinburgh, January 1989.

[29] T. A. Kean. Déjà vu, all over again. *IEEE Design and Test of Computers* 22(2), March/April 2005.

[30] J. T. McHenry, R. L. Donaldson. WILDFIRE custom configurable computer. *Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, Proceedings of the International Society of Optical Engineering*, October 1995.

[31] T. Miyazaki, T. Murooka, M. Katayama, A. Takahara. Transmutable telecom system and its application. *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1999.

[32] T. Miyazaki, K. Shirakawa, M. Katayama, T. Murooka, A. Takahara. A transmutable telecom system. *Field-Programmable Logic: From FPGAs to Computing Paradigms*, Springer-Verlag, August/September 1998.

[33] M. Moe, H. Schmit, S. Copen Goldstein. Characterization and parameterization of a pipeline reconfigurable FPGA. *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.

[34] D. S. Poznanovic. Application development on the SRC Computers, Inc. systems. *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[35] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, R. Reed Taylor. PipeRench: A virtualized programmable datapath in 0.18 micron technology. *Proceedings of the IEEE Custom Integrated Circuits Conference*, 2002.

[36] Silicon Graphics, Inc. Extraordinary acceleration of workflows with reconfigurable application-specific computing from SGI (*http://www.sgi.com/pdfs/3721.pdf*), 2004.

CHAPTER 4

RECONFIGURATION MANAGEMENT

Katherine Compton
Department of Electrical and Computer Engineering
University of Wisconsin–Madison

The flexibility of reconfigurable devices allows them to be customized to a wide variety of applications. Even individual applications can benefit from reconfigurability by using the hardware to perform different tasks at different times. If not all of an application's configurations fit on the hardware simultaneously, they can be swapped in and out as needed. In some cases, the circuitry implemented on reconfigurable hardware can also be optimized based on specific runtime conditions, further improving system efficiency. The process of reconfiguring the hardware at runtime, whether to accelerate different applications or different parts of an individual application, is (unsurprisingly) called *runtime reconfiguration* (RTR).

Unfortunately, although RTR can increase hardware utilization, it can also introduce significant *reconfiguration overhead*. Reconfiguring the hardware, depending on its capacity and design, can be very time consuming. Modern high-end FPGAs can have tens of millions of configuration points, and writing this information can require on the order of hundreds of milliseconds [3, 54]. In a reconfigurable computing system, where the compute-intensive portions of applications are implemented on reconfigurable hardware, computation and reconfiguration are mutually exclusive operations. Thus, time spent reconfiguring is time lost in terms of application acceleration. Studies estimate that, in some cases, reconfiguration time alone occupies approximately 25 to 98 percent of the total execution time of a reconfigurable computing application [36, 42, 50, 51]. Therefore, management and minimization of reconfiguration overhead to maximize the performance of reconfigurable computing systems is essential.

We first discuss the process of reconfiguration in Section 4.1 and then present different configuration architectures, including those designed specifically to help reduce reconfiguration overhead, in Section 4.2. Section 4.3 discusses the different issues in and approaches to managing the reconfiguration process to minimize reconfiguration overhead and maximize the benefit of hardware acceleration. Section 4.4 focuses on techniques that specifically reduce the configuration transfer time when a reconfiguration is required. Finally, Section 4.5 discusses configuration encryption to maintain intellectual property security in reconfigurable computing systems.

4.1 RECONFIGURATION

In reconfigurable devices, such as field-programmable gate arrays (FPGAs), logic and routing resources are controlled by reprogrammable memory locations, such as SRAM or Flash RAM. Boolean values held in these memory bits control whether certain wires are connected and what functionality is implemented by a particular piece of logic. The process of loading the Boolean values into these memory locations is called *reconfiguration*. A specific sequence of 1s and 0s for particular memory locations in hardware defines a specific circuit and is called a *configuration* for a given hardware task. Runtime reconfiguration therefore involves reconfiguring the device (loading a new set of 1s and 0s) with a different configuration (a specific sequence of 1s and 0s) from the one previously loaded in the reconfigurable hardware (RH). The configurations themselves are created by CAD software based on both the circuit design to be implemented and the architecture of the implementing RH. The architectural information is required for the design tools to know which configuration bits control which resources and what effect a 1 has versus a 0 in each of the configuration bit locations.

Once generated by the CAD tools, configurations are generally stored in a memory structure external to the RH. In some cases, configurations are stored in main memory and a CPU acts as the go-between, transferring them from memory to the RH as needed. In other cases, configurations are stored in a programmable ROM and a *configuration controller* loads the data directly from the ROM in the RH, potentially at the request of a central processing unit (CPU). The configuration controller and the ROM may be incorporated into the same device, such as the specialized configuration controllers marketed by various FPGA companies [3, 55], or they may be part of a user-designed custom device. Figure 4.1 shows a block diagram of a system using a configuration controller triggered by a CPU to reconfigure the RH (in this case, an FPGA). The configuration controller essentially implements a finite-state machine (FSM) that, based on the configuration requested by the CPU, generates the sequence of addresses needed to read the appropriate data sequence for that configuration out of the ROM.

4.2 CONFIGURATION ARCHITECTURES

A configuration architecture is the underlying physical circuitry that loads configuration data during reconfiguration, and holds it at the correct locations. Configuration architectures can range from simple serial shift chains, as discussed in the next section, to addressable structures that can manipulate configuration information after it is loaded. Some researchers have developed methods to emulate more complex configuration architectures on existing commercial designs, using a combination of hardware and software to provide advanced configuration functionalities. These approaches are discussed in Section 4.3.4.

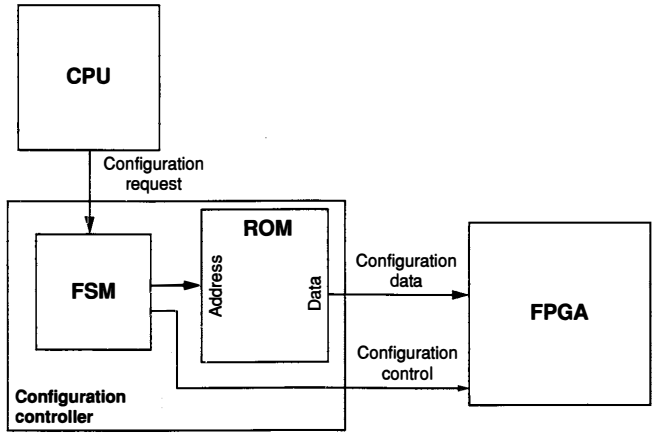


FIGURE 4.1 ■ Configuration data can be transferred to an FPGA by a specialized configuration controller containing nonvolatile ROM memory; the reconfiguration process can be triggered by a CPU.

4.2.1 Single-context

The single-context FPGA has been the most common choice in commercial designs, though there are exceptions. In this type of FPGA, configuration information is loaded into the programmable array through a serial shift chain, as shown in Figure 4.2.

Internally, the configuration architecture may actually be addressable, similar to a standard RAM device or the partially reconfigurable designs discussed in Section 4.2.3, but this would be an implementation detail hidden from the FPGA user. Addressable configuration architectures generally require fewer transistors per SRAM cell than serially programmed architectures, reducing the area required for configuration memory. In this case, an internal-state machine would control writing serially received data to locations in the array.

The Xilinx Virtex family of FPGAs have addressable configuration locations, but have a single-context configuration mode [54]. In these FPGAs, configuration data is divided up into addressable blocks called "frames," each of which corresponds to part of a column of reconfigurable resources. During reconfiguration, the configuration data is shifted into the frame data input register (FDRI) and from there written to a configuration memory location specified by the frame address register (FAR). For single-context configuration mode, this address starts at 0 and is automatically incremented each time a new frame is loaded. This allows the device to appear externally as a single-context device despite the addressability of the configuration information.

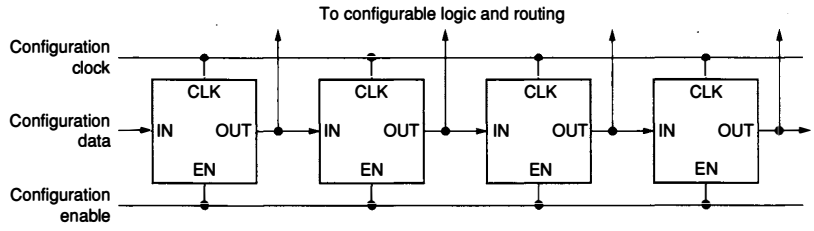


FIGURE 4.2 ■ Serially programmed FPGAs shift in configuration data. Each cell shown contains one SRAM bit of programming data. The clock controls shifting during configuration.

The benefit of serially programmed devices is that they require few pins for configuration, potentially simplifying board-level design. However, the entire chip must be reprogrammed for any change to the configuration data because the data cannot be selectively "reused" on the chip. For example, a large part of the structure of an encryption application may be independent of the chosen key, with only a relatively small portion optimized on a per-key basis. Ideally, only the key-dependent parts are reconfigured and the key-independent parts remain untouched when the key changes. However, a single-context design requires all configuration data to be rewritten during configuration, even if it is with the same values. A relatively minor change to the configuration data becomes a full reconfiguration process, replete with the associated delays.

The number of configuration cycles can be somewhat reduced in single-context devices by widening the configuration path. The Altera Stratix-II [3] and the Xilinx Virtex-II [54] receive either a single bit or a byte of configuration information per configuration clock cycle. The designer then chooses between the two modes by weighing the board-level design impact against the performance impact. As the larger Stratix II devices currently require more than 4MB of configuration data, with a maximum configuration clock speed of 100 MHz, the ability to configure in eight times fewer cycles can be significant. Newer Xilinx devices, such as the Virtex-5, allow a configuration data bus up to 32 bits wide [55].

4.2.2 Multi-context

For RTR systems, the overhead of serial programming may be prohibitive. An attractive alternative may be to provide storage in the device for multiple configurations simultaneously, facilitating configuration prefetching and fast reconfiguration. A *multi-context* device (sometimes called "time-multiplexed") contains multiple planes (contexts) of configuration data. Each configuration point of the device is controlled by a multiplexer that chooses between the context planes. Two configuration points for a 4-context device are shown in Figure 4.3. Several time-multiplexed FPGA architectures have been proposed, including Time-Multiplexed [47], DPGA [17], Dharma [11], and Morphosys [45].

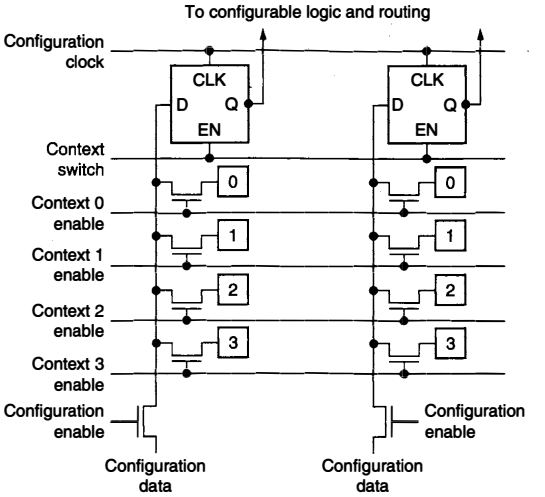


FIGURE 4.3 ■ Two multi-contexted configuration bits of a 4-context device.

Multi-context devices have two main benefits over single-context devices. First, they permit background loading of configuration data during circuit operation, overlapping computation with reconfiguration. Second, they can switch between stored configurations quickly—some in a single clock cycle—dramatically reducing reconfiguration overhead if the next configuration is present in one of the alternate contexts. However, if the next needed configuration is not present, there is still a significant penalty while the data is loaded. For that reason, either all needed contexts must fit in the available hardware or some control must determine when contexts should be loaded in order to minimize the number of wasted cycles stalling while reconfiguration completes. This type of control is discussed in Section 4.3.2.

One of the drawbacks of multi-contexted architectures is that the additional configuration data and required multiplexing occupies valuable area that could otherwise be used for logic or routing. Therefore, although multi-contexting can facilitate the use of an FPGA as virtual hardware, the *physical* capacity of a multi-contexted FPGA device is less than that of a single-context device of the same area. For example, a 4-context device has only 80 percent of the "active area" (simultaneously usable logic/routing resources) that a single-context device occupying the same fixed silicon area has [17]. A multi-context device limited to one active and one inactive context (a single SRAM plus a flip-flop) would have the advantages of background loading and fast context switching coupled with a lower area overhead, but it may not be appropriate if several different contexts are frequently reused.

Another drawback of multi-contexted devices is a direct consequence of its ability to perform a reconfiguration of the full device in a single cycle: spikes in dynamic power consumption. All configuration points are loaded from context memory simultaneously, and potentially the majority of configuration locations may be changed from 0 to 1 or vice versa. Switching many locations in a single cycle results in a significant momentary increase in dynamic power, which may violate system power constraints.

Finally, if any state-storing component of the FPGA is not connected to the configuration information, as may be true for flip-flops, its state will not be restored when switching back to the previous context. However, this issue can also be seen as a feature because it facilitates communication between configurations in other contexts by leaving partial results in place across configurations [27].

4.2.3 Partially Reconfigurable

Because not all configurations require the entire chip area, we might reduce reconfiguration time if we reloaded data only to those areas that actually must change. In partially reconfigurable devices, the configuration memory is addressable, similar to traditional RAM structures. If configurations are smaller than the full device, partial reconfiguration can decrease reconfiguration time by limiting reconfiguration to the resources used by a given configuration and, therefore the amount of configuration data to transfer. Partial reconfiguration can also allow multiple independent configurations to be swapped in and out of hardware independently, as one configuration can be selectively replaced on the chip while another is left intact. Furthermore, we can leverage the addressability to modify only part of a configuration already located on the chip if some of its structure matches a new configuration that we wish to load. For example, in an encryption circuit the bulk of the configuration may remain the same when the key is changed, and only a few resources may need to change based on the new key value. Partial reconfiguration can allow the system to reconfigure only those changed resources instead of the full circuit.

The Xilinx 6200 FPGA [53] was an early partially reconfigurable device where each logic block could be programmed individually. It therefore became a platform for a great deal of study of configuration architectures and RTR. Current partially reconfigurable commercial FPGAs include the Atmel AT40K [5] and the Xilinx Virtex FPGA family [54, 55]. The Virtex series is more coarsely reconfigurable than the 6200. Instead of addressing each logic block independently, reconfigures logic blocks in groups called frames. In the Virtex-II, a frame corresponds to part of a full column of resources and the size of the frame increases with the number of logic block rows in the device. In the Virtex 5, frames are fixed size of 41 32-bit words (regardless of device size) that represent a partial column of resources.

Although partially reconfigurable designs provide a great deal more flexibility for RTR systems, they can still stuffer from potential problems. First, if configurations occupy large areas of the device, the time saved transmitting configuration data may be outweighed by the time spent transmitting configuration addresses

In this case, a serially programmed FPGA may be more appropriate. Second, and more critical to RTR systems, partial configurations are generally fixed to specific locations on the device. If two independent configurations are implemented in overlapping hardware locations, they cannot operate simultaneously. One method of mitigating this issue is to view configuration placement as a three-dimensional floorplanning problem, with the third dimension representing time [6]. Configurations then occupy some three-dimensional volume of space based on physical location and time of use, allowing the floorplanner to determine the best two-dimensional placement to avoid time-related (three-dimensional) conflicts. Unfortunately, this technique cannot guarantee nonoverlapping configurations if the full configuration sequence is not known at compile time—a major problem in multitasking systems. The next section discusses advanced configuration architectures that eliminate configuration placement conflicts.

4.2.4 Relocation and Defragmentation

As previously discussed, conflicts between configuration locations can limit the effectiveness of partially reconfigurable architectures. To remove these conflicts, configurations should not be associated with fixed device locations. Relocation is a technique permitting configurations to be moved to different compatible device locations within the array, based where free area is available. Figure 4.4(a) shows a device loaded with configurations A, B, and C in sequence, each assigned to a free area. Figure 4.4(b) shows configurations A and B removed, and configuration D relocated and programmed onto the array.

The composition of the reconfigurable hardware can complicate this process in three critical ways. First, if the device's logic or routing is heterogeneous, relocation becomes less flexible, or even impossible, as a configuration may require resources located in only one or a few array locations. For example, in devices with hierarchical routing, different routing connections are available at different locations in the array. However, if heterogeneity is restricted to a repeating pattern, configurations can be relocated distances corresponding to some multiple of the distance of the repeat. To the relocated configuration, resources will be located in the same relative position as in the original placement.

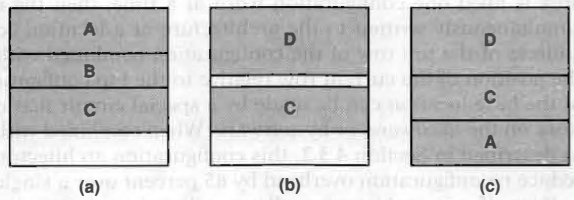


FIGURE 4.4 ■ Three configurations have been programmed on the hardware (a). In (b), A and B have been removed, and D has been relocated/configured to an available area, causing fragmentation. Defragmentation relocates configuration C to make room for configuration A when it is again needed, this time to a new location in the array (c).

Second, the external pin connections to the reconfigurable hardware fabric are fixed either at fabrication (reconfigurable hardware cores in a system-on-a-chip) or at board-level design (discrete FPGA components). If a configuration is relocated, its connections to the required I/O pins must be rerouted to maintain the proper connections. One solution to this problem is to use a communication network that itself has fixed pin connections but provides internal interfaces at multiple array locations to allow configurations to have the same communication connections regardless of position [14, 50]. This type of structure is known as *virtualized I/O* and can in some cases be emulated by using reconfigurable resources to implement a static communication structure and including the communication interfaces in individual dynamic configurations [7]. However, configurations must still be relocated such that they can still connect to the communication bus.

Third, a two-dimensional architecture can exacerbate the previous two problems, but particularly complicating virtualized I/O. If a configuration can be relocated both horizontally and vertically, the virtualized I/O must potentially distribute signals to all locations in the array. Furthermore, a two-dimensional architecture increases the possibilities for relocation, as we can consider not only configuration shifting but also rotation, which requires manipulating configuration information related to routing [14]. More relocation possibilities leads to a more complex relocation process and possibly increased configuration overhead.

A partially reconfigurable architecture designed specifically with relocation support should therefore require a homogeneous logic architecture, a bus-based communication structure, and a one-dimensional organization to simplify the relocation process [31, 50]. The one-dimensional architecture means that a configuration must use complete rows, even if it only needs a portion of a row. As device sizes increase, using rows as atomic reconfiguration units may become inefficient. Instead, the fabric can be split into multiple one-dimensional fabrics to retain the relocation benefits while preserving a reasonably sized atomic unit. The Virtex-5 device uses this approach [55].

One of the architectures designed for relocation [14] uses a "staging area" equivalent in size to one row of configuration data, which is similar in approach to the column-wise frame-based configuration method of the Xilinx Virtex family introduced in Section 4.2.1 and discussed in Section 4.2.3 [54]. The staging area is filled one configuration word at a time; then the entire row of data is simultaneously written to the architecture at a location computed with a base address of the top row of the configuration combined with an offset indicating the position of the current row relative to the top configuration row. The choice of the base location can be made by a special circuit that monitors empty locations on the hardware, or by software. When combined with the proper software as described in Section 4.3.2, this configuration architecture has been shown to reduce reconfiguration overhead by 85 percent over a single-context device [31].

Even if an architecture allows relocation, fragmentation of the usable resources can decrease its effectiveness. Like memory fragmentation, swapping configurations in and out of different places in the hardware can result in a situation where various locations in the array may be unused, but there may not

be enough contiguous space available to load a configuration. In this case, if the configurations can be defragmented, the new configuration can be loaded into the array without having to remove any of the configurations already on the device. Figure 4.4(b) shows an example of an array that has become fragmented, and Figure 4.4(c) shows how defragmentation can allow a configuration to be configured without having to remove an existing one. A simple approach to this problem is to remove all configurations, then reconfigure the array with the removed ones, this time relocating them to contiguous locations to eliminate fragmentation. However, this process involves significant communication overhead between fabric and configuration memory. Alternately, the reconfigurable hardware can move configurations internally, avoiding the need to communicate with configuration memory. The R/D FPGA [14, 31] provides both relocation and defragmentation ability, which together provide a 90 percent reduction in reconfiguration overhead compared to a single-context FPGA.

A configuration controller for one-dimensional hardware, such as the R/D FPGA, that specifically supports relocation and defragmentation may simply need to keep track of occupied and unoccupied locations, or request this information as needed from the hardware itself. The controller can determine locations for incoming configurations using a first-fit or best-fit method, similar to general memory allocation [7, 14]. Defragmentation, which is easy for the one-dimensional case, can be triggered when sufficient free area is available but is broken up into fragments too small to fit an incoming configuration. If there is insufficient free area, one or more configurations can be removed to make room, as described in Section 4.3.2.

4.2.5 Pipeline Reconfigurable

Pipeline reconfigurable arrays use a series of physical pipeline stages to implement the virtual pipeline stages of configurations. A virtual pipeline stage can be relocated to any physical pipeline stage, and the number of virtual stages is generally not constrained by the number of physical stages. The most well-known pipeline reconfigurable architecture is PipeRench [19], which is designed to implement deeply pipelined configurations, subdivided into a set of virtual pipeline stages. At runtime, the virtual pipeline stages are assigned to physical pipeline stage computation units. These units are arranged in a unidirectional ring, as shown in Figure 4.5(a). Although pipeline stages may be implemented in different physical locations over time, the virtual pipeline *appears* fixed to its own pipeline stages, with each stage receiving input from its predecessor and generating output to its successor. PipeRench permits pipeline stages to be configured in a single cycle to speed execution.

Pipeline reconfiguration eliminates many of the difficulties of using reconfigurable hardware as virtual hardware, but places restrictions on the circuits that can be implemented as information can only propagate forward through the pipeline stages, and any feedback connections must be completely contained within a single stage. Figure 4.5(b) shows a 4-stage virtual pipeline implemented on a 3-stage physical architecture.

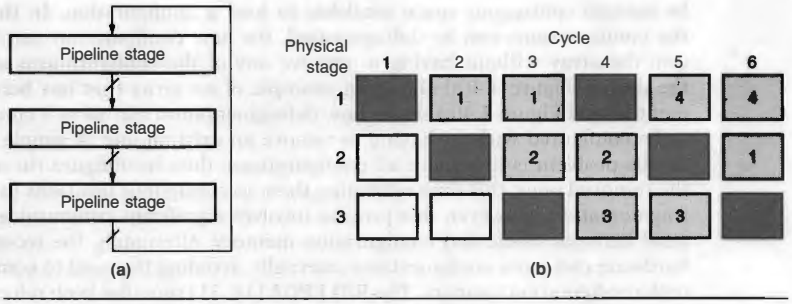


FIGURE 4.5 ■ A pipeline reconfigurable architecture with three physical stages (a). A 3-stage physical pipeline implementing a 4-stage virtual pipeline (b). Numbers within physical pipeline stages indicate the implemented virtual pipeline stage. Shaded stages are reconfiguring for the given cycle.

4.2.6 Block Reconfigurable

Block reconfigurable arrays can share characteristics with any of the previously described configuration architectures. However, rather than providing one large reconfigurable fabric, they are made up of multiple discrete blocks that can be used independently. For these purposes, "block" should not be confused with "logic block" in an FPGA. In this case each independent block can contain many logic resources. An individual configuration may occupy one or more blocks, but blocks may not be subdivided between configurations. Blocks are connected either through a crossbar structure [39] or a bus/network [10], as shown in Figure 4.6. Although this would seem to describe any architecture formed from multiple connected FPGAs or FPGA cores, block reconfigurable devices have the ability to relocate configurations to different blocks at runtime. For this reason, the blocks of reconfigurable logic in this style of architecture have also been referred to as "swappable logic units" (SLU) [55]. In the SLU architecture, a block reconfigurable design is implemented as an abstraction layer on top of a partially reconfigurable architecture to facilitate runtime relocation.

The SCORE reconfigurable architecture model [10] is a block reconfigurable design where the reconfigurable blocks are referred to as "pages" to evoke a virtual memory view of the reconfigurable hardware. Any virtual page can be implemented on any physical page, and computation pages are loaded as needed. Once configured, pages communicate with one another using datastreams over a scalable hierarchical network.

A heterogeneous multiprocessor may fit the block reconfigurable model, provided multiple blocks of reconfigurable hardware are present and configurations can be relocated between the blocks for computational flexibility. These architectures may contain a single communication network used by the configurable blocks and other resources such as microprocessors and custom circuitry. Although the Pleiades reconfigurable architecture [1] has some of these features (a heterogeneous multiprocessor with multiple reconfigurable blocks),

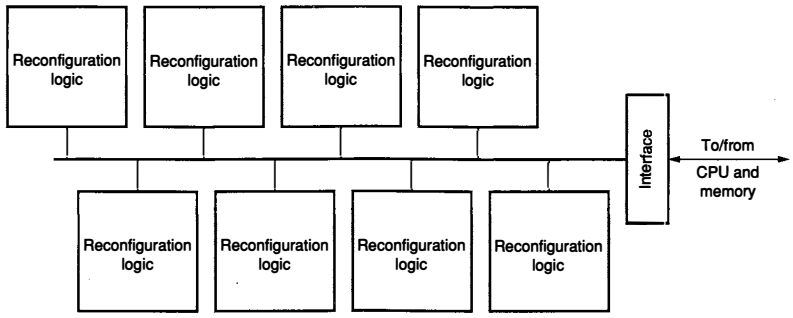


FIGURE 4.6 ■ In a block reconfigurable device, configurations can be relocated to any of the interconnected and equivalent blocks of reconfigurable logic.

computations are preassigned to specific resources, violating one of the requirements of the block reconfigurable category.

4.2.7 Summary

This section presented a variety of configuration architectures, each optimized for a different type of reconfiguration. The single-context device is the simplest in terms of configuration process and interface, and it is the most popular for current commercial devices. Partial reconfiguration, which allows reconfiguration of parts of the device (leaving the rest untouched), can reduce the amount of configuration data that must be transferred but is hampered by configuration placement conflicts. Partially reconfigurable designs augmented with relocation and defragmentation, as well as block reconfigurable designs, avoid this issue by allowing configurations to be placed at different locations from the ones originally assigned. Likewise, pipeline reconfigurable devices allow pipeline stages to be relocated but prohibit interstage feedback connections. Finally, multi-contexted devices provide a method for single-cycle device reconfiguration but at the cost of decreased computation resources for a given area and a dramatic increase in power consumption during context changes.

The more advanced reconfiguration architectures, such as relocation, defragmentation, and multi-contexting, have been popular for some time in the research community as tools essential for effective reconfigurable computing systems. However, such devices have not yet gained a significant market foothold because of the limited demand for fast reconfiguration capabilities. Instead, most FPGAs are currently used as drop-in ASIC replacements or as infrequently reconfigured hardware modified only for firmware updates. To provide devices at a competitive cost, most FPGA vendors forgo the more innovated configuration architectures in favor of a simpler single-context design. Although Xilinx, one of the most prominent FPGA vendors, offers partial reconfiguration in its Virtex families, design support is still somewhat limited, relocation

and defragmentation are not supported, and a single-context interface is still provided to cater to users who do not require partial reconfiguration. Even so, as reconfigurable computing becomes a more common practice, spurred perhaps by the difficulty of continued clock speed increases for general-purpose processors, demand for innovative configuration architectures will increase in order to maximize the benefits of reconfigurable computing.

4.3 MANAGING THE RECONFIGURATION PROCESS

Reconfigurable computing systems swap configurations in and out of hardware at runtime, a process controlled by software, hardware, or a combination of both. Although a system can simply load a configuration whenever it is needed, and unload it when hardware execution is complete, this can cause a significant reconfiguration overhead: while the configuration is loading, the controlling application or thread cannot compute. Also, if the hardware is currently in use by another thread or process, the requesting application or thread must wait until the hardware is idle or until enough area is free to even begin the reconfiguration process, leading to further stalling. Ideally, configurations are loaded in advance of when they are needed and those likely to be reused in the near future should be cached on the hardware.

The following sections discuss several aspects of reconfiguration control, including choosing the configurations to load, and when and where on the hardware to load them.

4.3.1 Configuration Grouping

Single-context and multi-context FPGAs may have more resources available at once than are usable by a single configuration. Reconfiguration overhead can be reduced by grouping configurations that are likely to be used one after another into a single larger configuration. Algorithms proposed to perform this grouping include simulated annealing and a clustering approach [31]. They examine the overall application control flow to predict configurations that should be grouped together. The loading of a grouped configuration involves not only the currently needed configuration but also those most likely to be used after. Therefore, if the next configuration requested is already present on the device, no reconfiguration is necessary, reducing reconfiguration overhead. With configuration grouping, a configuration will appear in at least one group, and possibly several, depending on application behavior and the configuration's relationship to other configurations.

This approach is primarily appropriate for single-application systems, as configuration grouping is a compile-time operation. However, it could also be used in a multitasking system with a multi-context device. In this case, the configuration grouping would still be performed at compile time for individual applications, and the choice of which configuration groups to load and when would be a runtime operation, as described in the next section.

4.3.2 Configuration Caching

In a single-context device, the loading of one configuration overwrites all configuration data in the FPGA. Thus, context grouping implicitly decides what operations will coexist within the device at any point. In a multi-context or partially reconfigurable architecture, reconfiguration only overwrites a portion of the configuration data, allowing other configurations to be retained elsewhere. With configuration caching, the goal is to keep configurations on the hardware if they are likely to be reused in the near future. If there is enough free area on the device to fit a requested configuration, it is simply loaded, but if there is insufficient space, the configuration controller must select one or more "victim" configurations to remove from the hardware to free the required area. This process is simplified from the point of view of the controller if the device does not support relocation, as the victim configurations are simply any that overlap with the incoming one. However, this will generally result in a high reconfiguration overhead, as the removed configurations could be needed again in the near future, requiring another reconfiguration.

If the device supports relocation and defragmentation, or multiple contexts, the controller may have a variety of potential victims to choose from that will free the needed area. In some cases, general caching approaches may be used. These approaches assume a fixed-sized data block. However, in a partially reconfigurable device the size of the block to load can vary because configurations can each use differing amounts of resources. The caching algorithm must therefore consider the impact of variable-sized blocks.

One algorithm uses a penalty-based approach that considers both the configuration's size and how recently it was used [31]. When a configuration is first loaded, its "credit" is set to its size. When one or more configurations must be removed to make room for an incoming one, the configuration with the lowest credit is chosen, and the credit values of the remaining configurations are lowered by the credit value of the removed one. For the R/D FPGA design [14], penalty-based caching consistently results in a lower reconfiguration overhead than a simple least recently used (LRU) approach and 90 percent less overhead than a single-context configuration architecture. A configuration controller for a multi-context device must select which context to overwrite when a new context not already in the device is requested [14]. Because each context is the same size, general caching techniques, such as LRU, have been used.

4.3.3 Configuration Scheduling

Configurations can be loaded simply as they are requested, but this may result in significant overhead if the software stalls while waiting for reconfiguration to complete [50]. If instead the system can request configurations in advance of when they are needed, a process called *prefetching*, reconfiguration may proceed concurrent with software execution until the hardware is actually required. The challenge, however, is to ensure that prefetched configurations will not be ejected from the hardware by other prefetching operations before they can be used. For example, Figure 4.7 shows a flow graph for an application containing both

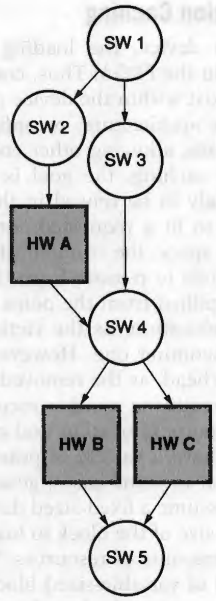


FIGURE 4.7 ■ An example reconfigurable computing application flow graph, containing both hardware and software components.

hardware and software components. Configuration A can safely begin loading at the beginning of the flow graph, provided that the application represented by the flow graph is the only one using the reconfigurable hardware. On the other hand, after the first branch rejoins at software block 4, it is unclear whether configuration B or configuration C will be needed next. If both potential branches have equal probability, the next configuration should not be loaded until after program flow determines the correct branch.

For static scheduling, prefetching commands may be inserted by the compiler based on static analysis of the application flow graph [23], and have been shown to reduce reconfiguration overhead up to a factor of 2. A more dynamic approach uses a Markov model to predict the next configuration that will be needed for a partially reconfigurable architecture with relocation and defragmentation [33]. Combining this approach with configuration caching results in a reconfiguration overhead reduction of a factor of 2 over configuration caching alone. Adding compiler "hints" to dynamic prediction achieves still better results.

Some dynamic approaches use the dataflow graph to determine when a given configuration is valid for execution [37, 39]. In these cases, nodes of the flow graph may be scheduled only if their ancestors have completed execution. This

approach works even if multiple applications are executing concurrently in the system and also works in systems implementing hardware tasks as independent "hardware threads" [4, 43].

Other approaches do not consider the actual flow graphs of applications, but instead use system status and current resource demand to allocate reconfigurable hardware to different configurations over time. Window-based scheduling periodically chooses the configurations to be implemented in hardware for the next "window" of time. This approach treats scheduling as a series of static problems yet still accommodates dynamic system behavior. One window-based scheduler uses a multi-constraint knapsack approach to choose configurations providing the best benefit (speedup) to the system as a whole based on configuration requests in the past window period. This technique was shown to increase overall *system* throughput by at least 20 percent relative to a processor without reconfigurable hardware [57].

In true multitasking systems load may not be consistent, with demand for the reconfigurable resources varying over time. This has led to more complex scheduling techniques that also consider modifying configurations based on available resources to take advantage of numerous resources when possible or to fit in limited resources when necessary [37, 40, 41, 57]. Another possibility is to permit a software alternative for configurations to avoid stalls if the hardware resources are in high demand [16, 34, 41, 57]. This approach allows dynamic binding of computations to hardware or software, where only the most beneficial configurations are actually implemented in hardware. Real-time systems similarly must choose tasks at runtime for hardware implementation based on real-time requirements (task priority, arrival and execution time, and deadlines), rejecting remaining tasks to software or possibly dropping them entirely [46].

4.3.4 Software-based Relocation and Defragmentation

Systems that do not support relocation and defragmentation at the configuration architecture level may support it at the software level to gain some of the associated benefits. However, this can be computationally intense for two-dimensional architectures. Finding a possible location for an arbitrarily shaped configuration can require an exhaustive search, which may incur a greater overhead penalty than the configuration penalty it seeks to avoid. Restricting configurations to rectangular shapes simplifies the process somewhat, though it is still a two-dimensional bin-packing problem. One approach to solving this problem is to maintain a list of empty spaces in the device and search it whenever a new configuration is to be loaded [6, 21, 48]. In either case, when the controller removes a configuration from the hardware, it can update the list based on the freed area. The "best" empty location to implement the incoming configuration can be chosen based on algorithms similar to one-dimensional packing, such as first-fit or best-fit.

When there are no empty locations that can fit the incoming configuration, the configuration controller can defragment the hardware to consolidate empty

space, or remove an existing configuration. Like two-dimensional relocation, two-dimensional defragmentation is very complex. It can be implemented by removing all configurations from the hardware and then successively reloading each one using one of the two-dimensional relocation techniques described previously. Alternately, a reconfiguration controller can use a technique specifically designed for two-dimensional defragmentation that rearranges only a subset of configurations, and dynamically schedules their movements in an effort to minimize disruption of those in execution [18].

A critical problem in supporting relocation, whether for the one-dimensional or the two-dimensional case, is rerouting the connections between a relocated configuration and the (nonrelocated) I/O pins. As discussed in Section 4.2.4, a virtualized I/O structure simplifies this problem, though virtualized I/O for two-dimensional architectures may be infeasibly large. However, if the architecture does not have virtualized I/O, either these signals must be rerouted at runtime [49] or the configurations must be modified to emulate virtualized I/O by having a specific movable interface to a nonrelocatable communications structure [7].

4.3.5 Context Switching

Unfortunately, some of the same terminology in the reconfigurable computing area is used to refer to different concepts. In this section, "context switch" does not refer to switching between planes of configuration data in a multi-context device. Instead, it refers to the suspend/resume behavior of processors (and potentially their associated reconfigurable logic) when multitasking. A few studies have discussed supporting suspend/resume of hardware operations as a way to support hardware multitasking [24, 44]. In these systems, long-running configurations may be interrupted to allow other configurations to proceed, and later be resumed to complete computation. Although the configuration state can be resumed by reloading the required configuration, the flip-flop values and the values stored in embedded RAM blocks are not necessarily part of the configuration, and therefore may require additional steps to save their state.

Reconfigurable hardware context switches may mirror processor context switches to facilitate hardware control by ensuring that the "owning" process is active and ready to receive results. The host processor may stall or wait while the reconfigurable hardware is active [43], or it may continue with parallel operations that are not dependent on the hardware's results [1, 24, 43].

4.4 REDUCING CONFIGURATION TRANSFER TIME

The various techniques described previously can reduce the number of times we have to reconfigure the hardware, or attempt to hide the configuration latency, but the actual time required to transfer a given configuration can also be reduced. One hardware-based technique already discussed in Section 4.2.3, partial reconfiguration, permits configuring only those parts of the hardware that are needed. The remainder of the chip does not need to be configured, and therefore

configuration data for these other areas does not need to be transferred. The next few sections present a number of other methods used to reduce the configuration transfer time, in most cases by reducing the amount of data transferred.

4.4.1 Architectural Approaches

The design of the reconfigurable architecture itself can affect the time required to configure it. For example, a coarse-grained architecture containing primarily fixed functional units will generally require fewer configuration bits for the same functionality than does a fine-grained LUT-based architecture [25]. Another architectural design feature that can impact reconfiguration times is the width of the configuration path. Section 4.2.1 discussed how a serially programmed FPGA can be programmed 8× faster if configuration data is loaded a byte per cycle instead of a bit per cycle. In cases where the reconfigurable hardware is located on the same chip as the configuration memory, a very wide path between them may be possible, drastically reducing reconfiguration time. For example, the R/D architecture [14] can have a wide enough path to an on-chip configuration cache to allow the entire staging area to be loaded in a single cycle.

4.4.2 Configuration Compression

Compression is a widely used method in general-purpose computing and networking to reduce data transfer times by reducing the number of bits transferred. Compression can also reduce the amount of configuration data transmitted to reconfigurable hardware, leading to a corresponding decrease in reconfiguration time. The first proposed configuration compression technique [22] targeted the Xilinx 6200-series FPGA [53], which, as discussed in Section 4.2.3, is partially reconfigurable at a very fine-grained level, addressing individual logic cells by their row and column. The 6200 includes two "wildcard registers," equal in bit width to the row and column addresses, which act as masks on the configuration addresses. This allows one piece of configuration data to be written to more than one location. Essentially, 0s in the wildcard register retain the configuration address bits for those locations, whereas 1s indicate that all possible combinations of values in those specific locations should be addressed. By treating wildcard register value generation as a logic minimization problem, configuration data is compressed by an average factor of four for the Xilinx 6200 [22].

An expansion of these efforts exploits the fact that not all configuration bits in a logic cell are used by all configurations [30]. In many cases, a number of bits in a logic cell configuration can be considered "don't-care" values and can be programmed either with a 1 or a 0 without affecting the configuration's functionality. This allows configuration data to be manipulated to increase the achievable compression rates by about a factor of 2. Although the wildcarding and don't-care approaches are effective, they are specific to a discontinued architecture. More recent studies [15, 32] examine the use of a variety of standard compression techniques that achieve up to a compression factor of 4 on more modern architectures.

Configuration compression is not merely an academic pursuit. Both Altera's and Xilinx's design tools can generate compressed configurations [3, 55]. The compressed configurations are stored in a separate configuration controller that decompresses them as they are sent to the FPGA. However, this form of compression only reduces configuration storage requirements and does not decrease the size of configuration data sent to the FPGA. Compressed configurations can, however, be loaded directly onto Stratix-II devices in some configuration modes, and decompressed on the FPGA itself.

4.4.3 Configuration Data Reuse

At times, only a portion of a configuration must be updated, such as the key-specific hardware in an encryption configuration. Rather than resend the full configuration information, a partially reconfigurable device allows just the changed portions to be sent. Circuits can be designed specifically to use partial reconfiguration to customize them based on constant values not known until runtime [58]. However, even less directly related configurations may also have configuration data in common. Certain computation or communication patterns may be common to several configurations, such as the use of adder structures, emphasis on near-neighbor routing instead of long-distance routing, and the like [20]. Similarly, there may be "default" values for configuration bits for unused resources, and two configurations may have used as well as unused resources in common. These commonalities can decrease the amount of "new" configuration data required to implement the next configuration, particularly if configuration data reuse is a factor in the design of the configurations. The degree of similarity is increased with a decrease in the granularity of reconfiguration (there are fewer ways for small sets of bits to differ than for large sets to differ) and can result in a decrease in configuration data by approximately 35 to 40 percent [35].

4.5 CONFIGURATION SECURITY

In most of this book, we view the programmability of an FPGA as an inherent advantage that provides a circuit implementation platform or a multi-purpose acceleration engine. However, this flexibility also increases the potential for intellectual property theft compared to custom ASIC hardware. SRAM-based FPGAs (the focus of this chapter), have volatile configuration memory; to retain configuration data, a battery must provide a constant power supply to the configuration bits. This configuration data is stored in memory (RAM or a PROM) external to the FPGA, and is loaded into the FPGA at power-up. Someone monitoring the wires between these structures could capture the configuration data flowing from memory to the reconfigurable device. They could then duplicate the circuit simply by loading that data onto a new chip. Design firms that create FPGA-based hardware want to protect their work (which may have required significant design time) and prevent reverse-engineering of their designs.

To discourage their unauthorized copying, FPGA configurations can be watermarked with a special signature based on the circuit designer and the purchasing customer [28]. Of course, the design can still be copied and reverse-engineered, but the watermark can help identify the source of the unauthorized copies.

Design security can also be provided by encrypting configuration data to obscure the employed design techniques and/or functionality [26]. Many FPGA vendors now support configuration encryption with special on-chip decryption hardware. The Xilinx Virtex-II, for example, uses triple-key DES [54], and Altera's Stratix-II [3], Actel's ProASIC3 [2], and Lattice's ECP2 [29] all support 128-bit AES configuration encryption. In all cases, the keys are stored in the FPGA, and encrypted configurations may only be loaded if they were encrypted with the same key as that stored in the device. For a Virtex-II device, a battery must be attached to the proper pins to retain the key when the device is not powered. In contrast, the Stratix-II, ECP2, and ProASIC3 devices use nonvolatile memory for key storage, eliminating the need for a separate battery.

For systems that do not require runtime reconfiguration, the opportunity to copy a design can be reduced in end-products by not transmitting the configuration data on probeable wires. Antifuse and Flash FPGAs, based on nonvolatile configuration memory structures, inherently retain configuration data on-chip once configured, avoiding the need to transfer the information for systems not using runtime reconfiguration.

4.6 SUMMARY

The difficulty of clock speed increases and power consumption concerns motivate reconfigurable computing as an important technique to advance digital design, implementing compute-intensive application tasks in reconfigurable hardware. However, the performance and power penalty of reconfiguration has the real potential to overwhelm its benefits. This chapter discussed a variety of methods proposed and used to reduce and in some cases remove reconfiguration overhead, including various configuration architecture designs, scheduling and caching techniques, and ways to reduce the configuration data size.

In many cases, several approaches can be combined to further reduce the overhead. For example, relocation and defragmentation architectural features facilitate advanced configuration scheduling mechanisms that load configurations in advance of their use to minimize processor stall time during reconfiguration. Likewise, a configuration cache can be combined with a relocation- and defragmentation-enabled design that uses a staging area, providing a wide path to configuration memory to decrease transfer time. This in turn can be combined with wildcarding to allow multiple identical rows or columns to be configured simultaneously. Such combined methods allow reconfigurable computing system designers to effectively minimize reconfiguration overhead and to provide the full benefit of reconfigurable computing in future computing systems.

References

[1] A. Abnous, H. Zhang, M. Wan, G. Varghese, V. Prabhu, J. Rabaey. The Pleiades architecture. *Application of Programmable DSPs in Mobile Communications*, A. Gatherer, A. Auslander, eds., Wiley, 2002.

[2] Actel Corp. *ProASIC3 Flash Family FPGAs*. Actel Corp., Mountain View, CA, 2006.

[3] Altera, Inc. *Stratix-II™ Device Handbook, Volumes 1 and 2*, Altera, Inc., San Jose, 2005.

[4] D. Andrews, D. Niehaus, R. Jidin. Implementing the thread programming model on hybrid FPGA/CPU computational components. *Workshop on Embedded Processor Architectures of the International Symposium on Computer Architecture*, 2004.

[5] Atmel Corp. *AT40K Series FPGA Interactive Architecture Guide*. Atmel Corp., San Jose, 1999.

[6] K. Bazargan, R. Kastner, M. Sarrafzadeh. Fast template placement for reconfigurable computing systems. *IEEE Design and Test, Special Issue on Reconfigurable Computing* 17(1), 2000.

[7] G. Brebner, O. Diessel. Chip-based reconfigurable task management. *International Conference on Field Programmable Logic and Applications*, 2001.

[8] J. Burns, A. Donlin, J. Hogg, S. Singh, M. de Wit. A dynamic reconfiguration runtime system. *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[9] J. M. P. Cardoso, M. Weinhardt. From C programs to the Configure-Execute model. *Design, Automation, and Test in Europe*, 2003.

[10] E. Caspi, A. DeHon, J. Wawrzynek. A streaming multithreaded model. *Third Workshop on Media and Stream Processors*, 2001.

[11] D. Chang, M. Marek-Sadowska. Partitioning sequential circuits on dynamically reconfigurable FPGAs. *IEEE Transactions on Computers* 48(6), 1999.

[12] M. C.-T. Chao, G.-M. Wu, I.-H.-R. Jiang, Y.-W. Chang. A clustering- and probability-based approach for time-multiplexed FPGA partitioning. *IEEE/ACM International Conference on Computer-Aided Design*, 1999.

[13] M. M. Chu. *Dynamic Runtime Scheduler Support for SCORE*, Master's thesis, University of California, Berkeley, 2000.

[14] K. Compton, Z. Li, J. Cooley, S. Knol, S. Hauck. Configuration relocation and defragmentation for runtime reconfigurable systems. *IEEE Transactions on VLSI* 10(3), June 2002.

[15] A. Dandalis, V. K. Prasanna. Configuration compression for FPGA-based embedded systems. *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2001.

[16] M. Dales. Managing a reconfigurable processor in a general purpose workstation environment. *Conference on Design, Automation, and Test in Europe*, 2003.

[17] A. DeHon. DPGA utilization and application. *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 1996.

[18] O. Diessel, H. E. Gindy, M. Middendorf, H. Schmeck, B. Schmidt. Dynamic scheduling of tasks on partially reconfigurable FPGAs. *IEE Proceedings—Computers and Digital Techniques, Special Issue on Reconfigurable Systems* 147(3), 2000.

[19] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R. R. Taylor. PipeRench: A reconfigurable architecture and compiler. *IEEE Computer* 33(4), April 2000.

[20] J. D. Hadley, B. L. Hutchings. Design methodologies for partially reconfigured systems. *IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[21] M. Handa, R. Vemuri. An efficient algorithm for finding empty space for online FPGA placement. *Design Automation Conference*, 2004.
[22] S. Hauck, Z. Li, E. J. Schwabe. Configuration compression for the Xilinx XC6200 FPGA. *IEEE Symposium on FPGAs for Custom Computing Machines*, 1998.
[23] S. Hauck. Configuration prefetch for single context reconfigurable coprocessors. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 1998.
[24] J. R. Hauser. *Augmenting a Microprocessor with Reconfigurable Hardware*, Ph.D. thesis, University of California, Berkeley, 2000.
[25] Z. Huang, S. Malik. Managing dynamic reconfiguration overhead in systems-on-a-chip design using reconfigurable datapaths and optimized interconnection networks. *Design, Automation, and Test in Europe*, 2001.
[26] T. Kean. Cryptographic rights management of FPGA intellectual property cores. *International Symposium on Field-Programmable Gate Arrays*, 2002.
[27] A. Khan, N. Miyamoto, T. Ohkawa, A. Jamak, S. Kita, K. Kotani, T. Ohmi. An approach to realize time-sharing of flip-flops in time-multiplexed FPGAs. *IEEE International Conference on Field-Programmable Technology*, 2004.
[28] J. Lach, W. H. Mangione-Smith, M. Potkonjak. Fingerprinting techniques for field-programmable gate array intellectual property protection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20(10), 2001.
[29] Lattice Semiconductor Corp. *LatticeECP2 Family Data Sheet*, Lattice Semiconductor Corp., Hillsboro, OR, 2006.
[30] Z. Li, S. Hauck. Don't care discovery for FPGA configuration compression. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 1999.
[31] Z. Li, K. Compton, S. Hauck. Configuration caching for FPGAs. *IEEE Symposium on FPGAs for Custom Computing Machines*, 2000.
[32] Z. Li, S. Hauck. Configuration compression for Virtex FPGAs. *IEEE Symposium on FPGAs for Custom Computing Machines*, 2001.
[33] Z. Li, S. Hauck. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2002.
[34] R. Lysecky, F. Valid. A configurable logic architecture for dynamic hardware/software partitioning. *Design, Automation, and Test in Europe*, 2004.
[35] U. Malik, O. Diessel. On the placement and granularity of FPGA configurations. *IEEE International Conference on Field-Programmable Technology*, 2004.
[36] W. H. Mangione-Smith. ATR from UCLA. Personal communication, 1999.
[37] Y. Markovskiy, E. Caspi, R. Huang, J. Yeh, M. Chu, J. Wawrzynek, A. DeHon. Analysis of quasi-static scheduling techniques in a virtualized reconfigurable machine. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2002.
[38] J. Noguera, R. M. Badia. HW/SW codesign techniques for dynamically reconfigurable architectures. *IEEE Transactions on VLSI Systems* 10(4), 2002.
[39] J. Noguera, R. M. Badia. Multitasking on reconfigurable architectures: Microarchitecture support and dynamic scheduling. *ACM Transactions on Embedded Computing Systems* 3(2), May 2004.
[40] V. Nollet, P. Coene, D. Verkest, S. Vernalde, R. Lauwereins. Designing an operating system for a heterogeneous reconfigurable SoC. *Reconfigurable Architecture Workshop*, 2003.
[41] H. Quinn, L. S. King, M. Leeser, W. Meleis. Runtime assignment of reconfigurable hardware components for image processing pipelines. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003.

[42] J. Resano, D. Mozos, F. Catthoor. A hybrid prefetch scheduling heuristic to minimize at runtime the reconfiguration overhead of dynamically reconfigurable hardware. *Design, Automation, and Test in Europe*, 2005.

[43] C. R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. M. Arnold, M. Gokhale. The NAPA adaptive processing architecture. *IEEE Symposium on FPGAs for Custom Computing Machines*, 1998.

[44] H. Simmler, L. Levinson, R. Männer. Multitasking on FPGA coprocessors. *International Conference on Field-Programmable Logic and Applications*, 2000.

[45] H. Singh, G. Lu, M.-H. Lee, F. Kurdahi, N. Bagherzadeh, E. Filho, R. Mastre. MorphoSys: Case study of a reconfigurable computing system targeting multimedia applications. *Design Automation Conference*, 2000.

[46] C. Steiger, H. Walder, M. Platzner. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Transactions on Computers* 53(11), 2004.

[47] S. Trimberger, D. Carberry, A. Johnson, J. Wong. A time-multiplexed FPGA. *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[48] H. Walder, M. Platzner. Non-preemptive multitasking on FPGAs: Task placement and footprint transform. *International Conference on Engineering of Reconfigurable Systems and Architectures*, 2002.

[49] G. Wigley, D. Kearney. The first real operating system for reconfigurable computing. *Australasian Computer Systems Architecture Conference*, 2001.

[50] M. J. Wirthlin, B. L. Hutchings. A dynamic instruction set computer. *IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[51] M. J. Wirthlin, B. L. Hutchings. Sequencing run-time reconfigured hardware with software. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 1996.

[52] G.-M. Wu, J.-M. Lin, Y.-W. Chang. Generic ILP-based approaches for time-multiplexed FPGA partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20(10), 2001.

[53] Xilinx, Inc. *XC6200 Field Programmable Gate Arrays Product Description*, Xilinx, Inc., San Jose, 1997.

[54] Xilinx, Inc. *Virtex-II Platform FPGAs: Complete Data Sheet*, Xilinx, Inc., San Jose, 2004.

[55] Xilinx, Inc. *Virtex-5 FPGA Configuration User Guide*, Xilinx, Inc., San Jose, 2006.

[56] G. Brebner. The swappable logic unit: A paradigm for virtual hardware, *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[57] W. Fu, K. Compton. An execution environment for reconfigurable computing. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005.

[58] M. Wirthlin, B. Hutchings. Improving functional density through run-time constant propagation. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 86–92, 1997.

PART II

PROGRAMMING RECONFIGURABLE SYSTEMS

As suggested in the Introduction, field-programmable gate arrays (FPGAs) and reconfigurable architectures have both the postfabrication programmability of software and the spatial parallelism of hardware. To fully exploit them, we need models and programming approaches that support the software's programmability, including infrastructure and runtime support to allow the configuration to change over time. In addition to temporal reprogrammability, the reconfigurable programming systems must simultaneously deal with spatial issues normally associated only with hardware (e.g., physical placement of computations and timing of functional units).

To illustrate how we can program reconfigurable systems, the chapters in this part of the book describe the current state of the art in approaching and capturing designs for FPGAs and reconfigurable architectures. Chapter 5 reviews compute models and organizations suitable for reconfigurable applications, Chapters 6 through 10 and Chapter 12 explore different design entry points for reconfigurable applications, and Chapters 11 and 12 examine infrastructural support issues, including operating and runtime systems and debuggers.

The flexibility of FPGAs and reconfigurable architectures, as well as their dual hardware/software nature, means that the old computational models we are familiar with for hardware or software may not be the most effective for reasoning about reconfigurable designs. Furthermore, the design space for reconfigurable solutions is much larger than those most of us are used to navigating. Chapter 5 explores some useful models for capturing and conceptualizing reconfigurable applications and a variety of system architectures for providing efficient implementations. A clear conceptual model of the parallelism in the application, how to expose it, and how to exploit it make up an invaluable starting point for describing the application in a concrete programming language.

Chapter 6 provides an introduction to VHDL as an example of a Register transfer level (RTL) hardware description language. A software designer might think of VHDL as a semi-portable assembly language for reconfigurable designs; it provides fine control of hardware and

parallelism, but it demands that the designer manage quite a number of low-level details. Many of the higher-level programming approaches still use VHDL as an intermediate mapping stage on the way to a reconfigurable configuration.

Chapter 7 turns to more software-friendly approaches and shows how programs written in C can automatically be translated into reconfigurable hardware designs. Today, we cannot expect to obtain good performance from arbitrary C code with no concern for the capabilities of the reconfigurable architecture and compiler. However, with an appreciation for what reconfigurable architectures can do, an appropriate system architecture, and an understanding of the capabilities of the C compiler, it is possible to effectively develop and optimize reconfigurable applications in C.

Chapters 8 and 9 discuss two examples of programming systems that support streaming dataflow compute models (Section 5.1.3). These models, too, provide a higher-level approach to reconfigurable design than VHDL, offering greater opportunities for automated design scalability. Chapter 8 describes how we can apply the SDF (Synchronous Dataflow) model (Section 5.1.3) using Simulink, illustrating how methodology and suitable libraries can raise the abstraction for design construction. These techniques can readily be adopted by today's system designers. At the same time, the Simulink integration example shows how reconfigurable design can leverage popular system analysis tools such as MATLAB.

Chapter 9 describes a more custom and automated experimental design flow that supports application scalability for dynamic streaming dataflow applications (Section 5.1.3). It illustrates how many system architectures (Section 5.2) come together to support efficient and automated mapping of designs to reconfigurable computing platforms, and it offers a vision of how integrated programming systems for reconfigurable platforms might evolve.

Many efficient reconfigurable applications are naturally data parallel (Section 5.1.5) and are efficiently implemented with a Single Instruction Multiple Data (SIMD) or vector organization. Chapter 10 describes data parallel programming approaches customized for reconfigurable compilation.

In Chapter 12 we see an example of a rich generator language, JHDL, which provides even lower-level control of structure than VHDL, but does so with the full programming power of a conventional software language, Java. Thus, it provides a high-level platform from which to develop highly tuned designs. It also provides rich support for the construction of custom tools for reconfigurable design optimization.

As reconfigurable computers emerge as platforms for creating and delivering software, we must develop software support normally associated only with general-purpose processors, including operating systems, runtime support, and interactive debuggers. Chapter 11 describes the growing demands for reconfigurable operating systems, highlighting some of the early work along this path and pointing out important directions for the future. JHDL (Chapter 12) is notable for its support for interactive debugging and the extensible programming environment it provides, including hooks for software modules that interact with reconfigurable designs.

CHAPTER 5

COMPUTE MODELS AND SYSTEM ARCHITECTURES

André DeHon
Department of Electrical and Systems Engineering
University of Pennsylvania

Field-programmable gate array (FPGA) and reconfigurable architectures provide enormous raw computing power and tremendous flexibility. How do we best exploit this opportunity and bring it to bear on particular computing tasks? When we do take advantage of the flexibility, and how do we ensure correctness? How do we preserve and reuse our designs as technology continues to advance? The raw size and flexibility of today's devices and systems make these questions daunting to consider and intractable to approach in an undisciplined manner. In this chapter, we review models and organizational styles for large-scale, highly parallel computing resources and emphasize how they can be used in the organization of reconfigurable computers.

A modern FPGA has hundreds of thousands of independently configured bit-processing units and hundreds of memories. Today's multi-FPGA systems and future single-chip FPGAs raise these numbers to millions of bit-processing units and thousands or tens of thousands of memories. Furthermore, configurable interconnect allows us to arrange these resources in almost any manner. This gives us the power to adapt the computation to a particular task. Now that we have that power, what do we do with it?

Developing large software applications is a known hard problem, and managing resources and computations in highly parallel systems is, notoriously, even harder. Without care, our parallel computations may behave differently on each execution, producing nondeterministic results, some of which may be erroneous, and some executions may lead to deadlock. Unconstrained, the additional flexibility that comes with parallelism increases the complexity of application development and verification.

Considering both the limits of the human mind and the desire to achieve reasonably low time-to-solution periods, we cannot afford to custom-tailor each 4-LUT and each memory. With industry producing new devices according to Moore's Law, we cannot afford to design for 100,000 4-LUTs one year, discard the design, and then redesign for 200,000 4-LUTs three years later when the next part becomes available. Nor can we afford to reason about the interaction of every individual 4-LUT with every other—a number of interactions that grows quadratically with resource count.

The good news is that, while there is almost unbounded freedom in how we might solve problems, there are a small number of high-level organizational strategies that suffice to describe and efficiently implement most computing tasks. To bridge the semantic gap between applications and FPGA resources, we should think about two abstractions:

- *Compute models*—high-level models of the flow of computation in an application, useful for capturing parallelism and reasoning about correctness of implementations.
- *System architectures*—high-level strategies for organizing resources, managing the parallelism in the implementation, and facilitating optimization and design scaling.

Within each system architecture, there remains considerable flexibility to tailor the computing resources to the particular task, exploiting the flexibility of the architecture's reconfigurability. The compute model provides high-level constraints and guidance for conceptualizing the problem, reasoning about its correctness, and supporting manual and automated optimization. Chosen properly, the compute model naturally captures the parallelism of the application, making it easier to reason about its description and mapping.

A diversity of compute models and system architectures is needed to capture the diversity of natural organizations and implementations of tasks. Nonetheless, evidence to date suggests that there are only a modest number, perhaps tens of each, necessary to do this. Mismatches between the compute model and the task increase the complexity and awkwardness of the design and limit scalability. However, a good designers will be aware of the variety of compute models and system architectures and judiciously select the ones that naturally match her problem.

For decades, software engineers have faced the problem of managing complexity in large, highly concurrent software systems. *Software architectures* [1] were developed as one of the organizational tools to manage the complexity and to guide the design of these systems. The *system architectures* identified here are a deliberate expansion and adaptation of software architecture for reconfigurable computing, and many of the challenges are identical. However, the additional flexibility of reconfigurable architectures opens up design options and tradeoffs not typically present in the conventional multiprocessor systems for which software architectures have been traditionally targeted.

The two main sections in this chapter introduce, respectively, compute models and system architectures relevant to reconfigurable computing. For the reader approaching these topics for the first time, it may make sense to read the introductory sections, giving the detailed sections only a cursory review, for a high-level understanding of why we need a variety of models and architectures. As one delves further into reconfigurable designs or has a particular application in mind to solve, the in-depth sections can serve as a reference guide and provide deeper consideration of the merits and suitability of each approach.

5.1 COMPUTE MODELS

Figure 5.1 provides a taxonomy of the major compute models discussed and refined in this chapter. The leftmost branch is a set of models organized around the flow of data between operators; in these we think about the computation as a graph of computational operators and we reason about the correctness and assembly of operation in terms of data arrival at the operators, the function performed on the data, and the result produced and forwarded to other operators. The rightmost branch is a set of models organized around synchronous steps for the entire machine; here we think about the computation as a sequence of, perhaps parallel, operations performing transformation to global state.

At the top of the figure is a generic multi-threaded model or, formally, a model such as Hoare's Communicating Sequential Processes (CSP) [2]. All of the models below can be seen as refinements and stylizations on it. The multi-threaded model gives little guidance to the programmer on how to organize and design programs. Consequently, each of the refinements takes a stronger stand on how computation and parallelism are organized and how we manage synchronization. In many cases the refined models come with greater opportunities for optimization and stronger verification guarantees.

As we will see, system architectures are typically built on some of the same distinctions identified here in compute models (e.g., sequential control versus dataflow). However, there is not necessarily a one-to-one matching between the compute model used for capturing and reasoning about the application and the system architecture used for implementation. For example, modern superscalar microprocessors efficiently execute sequential instructions streams using dataflow techniques (e.g., Tomasulo [3]), and digital signal processors (DSPs) execute synchronous dataflow graphs as a sequence of instructions.

5.1.1 Challenges

When approaching a problem, we want to know how to implement the desired computation correctly, with the least effort, while exploiting the available

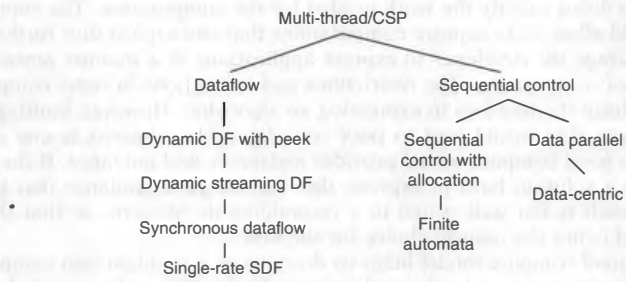


FIGURE 5.1 ■ Overview of compute models.

hardware capabilities on current and future machines. We can decompose this into specific challenges in selecting a compute model and developing the implementation:

- How do we think about composing the application?
- How does the compute model naturally lead to efficient, spatial solutions?
- How does the compute model support design (de)composition?
- How do we conceptualize parallelism?
- How do we trade area for time in the compute model?
- How do we reason about correctness?
- How do we deal with technology effects and adapt to technology changes?
- How does the compute model provide or guarantee determinacy?
- How do we avoid deadlock?
- What can we compute?
- How complex is it to optimize or validate properties of the application?

The first thing the compute model gives us is a way to think about the application. For example: Should we think of the application as a sequence of operations that need to be performed (sequential control)? As applying an operation to a set of independent data items (data parallel)? As a set of transformations on a data sequence (streaming dataflow)? To the extent these questions provide a natural way to describe the application, they make it easier to compose the application, identify the natural parallelism, and reason about correctness and transformations. Ideally, the compute model acts as part of the bridge between the application and the reconfigurable platform, providing a modest semantic gap between the application and the system architecture. The system architecture then brings to bear a large set of knowledge, accumulated across many applications, about how to efficiently bridge the gap between the compute model and the reconfigurable platform.

Reconfigurable platforms are most efficient when we can arrange for each resource to do the same thing over and over, and keep most of the resources active doing exactly the work needed for the computation. The compute model should allow us to capture computations that can exploit this; further, it should encourage the developer to express applications in a manner amenable to this kind of computation. The restrictions and stylizations in some compute models may limit the freedom in expressing an algorithm. However, limiting expressive freedom that would lead to poor reconfigurable solutions is one of the ways that a good compute model provides assistance and guidance. If the limitations make a solution hard to express, that can be good guidance that the solution approach is not well suited to a reconfigurable platform or that the compute model is not the natural choice for the task.

A good compute model helps us decompose a problem into components that can be designed and validated independently. This helps avoid the quadratic explosion in complexity arising from potentially interacting resources, and

even avoid the linear cost necessary if we had to program each resource independently. Sequential control models may focus on sequences of subroutines; dataflow models, on composition of functions; and streaming dataflow models, on hierarchical operator graphs.

Important to identify early is where the parallelism exists in an application. Is it between data items (data parallelism), between coarse-grained tasks (task parallelism), between operators in a task (instruction-level parallelism), or within low-level arithmetic and binary operations (bit-level parallelism)? Identifying and exposing these opportunities assists area-time tradeoffs: On small, economical platforms, we can tune a task for the modest area at the expense of longer runtime, while on larger platforms we might exploit the additional area to reduce compute time. Parallelism shows up implicitly or explicitly in each compute model, and a good match in parallelism will facilitate successful application scaling.

One of the most important tools provided by each compute model is a way to reason about correctness, which ultimately facilitates scaling, implementation adaptation, and optimization because it defines what transformations are possible without impacting correctness. In a sequential control model we identify the visible state on each step and reason about the changes in it; in a streaming dataflow model we reason about the output sequence of a computational graph.

With rapidly advancing technology, the size, speed, and energy of computing primitives (e.g., gates, wires, memories) are changing continually as they move from platform to platform. Sometimes they move together, with compute, interconnect, and memory speeds all growing uniformly smaller. Often, however, they change at different rates. As vendors have optimized memory for density and logic for speed, relative speeds have diverged, and, as we reach into the deep submicron regime, interconnect scales more slowly than compute. As a result, simply moving an old design to a new platform is unlikely to optimally exploit it. With increasing interconnect delays, perhaps the design needs more pipelining to distant locations; with slower memories, perhaps it needs more parallel memory blocks servicing a compute block. The compute model helps us understand the transformations permissible for the design, which may point to techniques the system architecture can employ for tolerating changes in constituent delays. Stall signals, for example, allow sequential control to slow down only when uncommon operations run at slower speeds than the scaled speed of the rest of the logic; data presence (see Data presence subsection of Section 5.2.1) allows streaming dataflow computations to tolerate variable delays within and between operators.

Given the same set of inputs, we might want our computation to produce the same outputs. That is, we often want our computation to be *deterministic*. Certainly, if the result of the computation differs each time it is performed, it becomes harder to debug our application or demonstrate its correctness. This can be a mild problem with sequential applications, where dependence on dynamic effects (e.g., dynamically allocated addresses) may change the program behavior; it becomes acute in concurrent systems. If there is variability in the relative

timing of operations, the order of events can change, and without care this may result in different visible application behavior.

Further, as we scale to different hardware capacities, we may exploit different amounts of concurrency and deliberately change the order of primitive events. Nonetheless, we might want to guarantee that the application remains deterministic, providing the same results for any legal parallelism. Some compute models with limited constraints may not be able to guarantee such determinacy but place this burden on the individual programmer. Most, however, come with disciplines that the developer can use to provide determinism, and some come with a sufficient set of model restrictions to automatically guarantee it.

Still, sometimes we want or need nondeterminism to deal with variations in the outside world (e.g., waiting for human input) or with deliberate variations to avoid bad behavior (e.g., randomized algorithms). Sometimes, too, there are multiple "correct" results and it is efficient to allow the system to select any of them, perhaps in a way that looks nondeterministic to the application as a whole. The point here is that nondeterminism always adds complexity to construction and validation, so it should be used sparingly and with care [4]

When dealing with shared or limited resources or variable operations in concurrent systems, we must also watch out for *deadlock*; in other words, we must watch for cases where the system may enter a state that prevents it from making forward progress. Often deadlock occurs when we attempt to give exclusive access to resources in an application. If a set of tasks end up waiting for each other—that is, the task set has a dependent cycle waiting for resources—the tasks can become deadlocked and the application will never complete. This can happen in purely deterministic computations, but should be at least identified by reasonably testing if the paths through the code are largely data independent. However, if the paths are largely data dependent, and deadlock only occurs for certain data values, identifying it with *ad hoc* testing can be difficult. When resource allocation and sequencing are nondeterministic, avoiding deadlock can be even more tricky. For these reasons, it is necessary to carefully guarantee that none of the legal, nondeterministic choices leads to a deadlock situation.

Computational theory gives us a well-developed set of models for computation. The Church–Turing Thesis [5–7] suggests that there is a very robust class of computing models that are all equivalent to the Turing Machine or the Lambda Calculus model. In fact, most of the models discussed here are Turing Complete. However, some refinements, such as synchronous dataflow (see Synchronous dataflow subsection of Section 5.1.3) or finite-state sequential control (see Finite state subsection of Section 5.1.4) models, are specifically less powerful. As will be noted, these restricted models give up expressive power in order to gain more powerful optimization and analysis.

We want to be able to say that an application always has certain properties. Ideally, we can verify that our expression of the application is correct, or, more specifically, that our captured algorithm is deterministic or that it can never deadlock. Further, to facilitate automated optimization and area–time scaling, we must guarantee that any changes made to the implementation preserve determinism and freedom from deadlock. Thus, we are ultimately concerned with the

computational tractability of verification and optimization. In Turing–Complete compute models, where anything is allowed, verification and general optimization can be *undecidable*; that is, without solving the *halting problem* it is not possible to analyze the design and say whether or not it is correct, determinate, or deadlock free. In more restricted models, verification or optimization may be *decidable* but *NP-hard*, meaning that we know of no polynomial time solutions to perform the optimization. And in even more restricted models, verification and optimization may be polynomial time. Consequently, we have a trade-off between the expressiveness and the strength of automation we can bring to bear on the problem, which suggests that the designer carefully select compute models that are expressive enough for her problem but not unnecessarily so.

5.1.2 Common Primitives

Two common primitives useful for defining and reasoning about compute models are *functions* and *objects*.

Function

A *function* is simply a deterministic, mathematical function that maps each finite input to a finite output:

$$Y = [y_0, y_1, \ldots, y_n] = f(X = [x_0, x_1, \ldots, x_m])$$

A function depends on no hidden state but only the input arguments to it, and it modifies no state values. Examples include addition, square root, and discrete-cosine transform (DCT). Functions can be composed, and the result is another function. For example:

$$y = (f \circ g)(x) = f(g(x))$$

Functions are interesting as a building block for several reasons:

- Functions are a useful formal primitive for defining computational models.
- Functional operations can be a tool or clue to parallelism—since functions do not modify state, they may be evaluated in parallel; evaluation of functions on different data can often be heavily pipelined.
- Functions can be a tool or guide to recurrent computations—those that show up regularly in the description of a computation are candidates for computational blocks that can be profitably implemented in spatial reconfigurable logic.

Transform or object

We can associate state with a function in order to create a common building block we can think of as a *transform*, or a primitive version of an *object*. In signal processing, we might think of a general transform as taking a sequence

of inputs and computing outputs based on them as well as on some finite state from the previous output:

$$Y_i = f(X_i, Y_{i-1})$$

In an *object-oriented* model, we might think of the object, O, being the combination of state, $O.s$, and a function, $O.f$, with each invocation evaluating the function on the input and the state and returning an output and a new state value:

$$Y, O.s_i = O.f(X, O.s_{i-1})$$

Examples of transforms include accumulators, finite-impulse response filters (FIRs), infinite impulse response filters (IIRs), and linear-feedback shift registers (LFSRs).

This primitive object or transform is more powerful than a pure function, but the inclusion of state may restrict its freedom of usage and implementation. As described, the state is finite, and each object can be viewed as a finite automata. The model says that the sequential invocations of an object see the state from the previous invocation; this demands that we complete the function's evaluation before starting the next invocation—or, at least, that we provide an implementation that produces the same net output sequence and state updates as though we had done so. For simple functions (e.g., LFSRs) or those where the state can be maintained without computation (e.g., FIRs), we can still pipeline the operation heavily. However, for complex functions (e.g., IIRs) the state feedback may limit our ability to heavily pipeline the object.

Nonetheless, object state is owned by the object, so evaluation of an object affects no others. Consequently, distinct objects with a complete set of inputs can evaluate in parallel; they impact each other only by communicating values between them. Further, objects with the same function may be able to share the same hardware to create commonality. This is useful both for enabling area–time trade-offs and for keeping a spatial datapath active in repeatedly performing the same operations. If sequential dependencies within an operator limit pipelining and we have many objects of the same type, it may be possible to C-slow the function evaluation (Chapter 18) to use the same hardware to service multiple objects.

In rich object-oriented models, we may associate additional capabilities with objects. We will introduce some of these as we explore more powerful compute models in the following sections.

5.1.3 Dataflow

We begin our detailed discussion of compute models with the left branch in Figure 5.1. In these models, we reason about the computation based on the flow of data. Computations are performed by *operators*, which can be either functions or objects as defined previously. We connect the operators into a graph, linking the output data from one to the input data of another. When its inputs arrive,

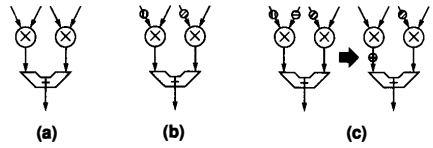


FIGURE 5.2 ■ Computation on a dataflow graph: (a) graph without inputs, (b) graph with partial inputs, and (c) arrival of matched input on left ×-operator allows it to evaluate and compute its output.

an operator can evaluate, produce its outputs, and send them to any operators connected to it (see Figure 5.2).

The dataflow graph exposes considerable parallelism and freedom in evaluation permitted to an implementation. The links capture the communication and dependence structure of the computation explicitly.

There is a large hierarchy of dataflow models with different flexibilities and challenges. For example, the simple models can be easily mapped to spatial, reconfigurable computation. The more flexible and powerful models are more complicated to implement efficiently, and make it difficult to guarantee correctness. However, for some applications, these more powerful models may be essential to efficiently describing and executing an application.

Single-rate synchronous dataflow

One of the most primitive dataflow models is that of a static graph of operators. The graph is created once, before the application executes, and persists unchanged throughout execution. In contrast, in the Streaming dataflow with allocation subsection (see page 102), we will consider models that allow the dataflow graph to change as part of the computation. We call the persistent edges between operators *streams* or *pipes*, as they deliver a sequence of values from a single producer to a single consumer, and we identify each value carried over these streams as a *token*. Such a graph of operators can itself be viewed as an operator, so this provides a model for composition of more powerful operators from more primitive functions and objects (see Figure 5.3). Computationally, this still provides the power of a finite automata, but the dataflow view is often a more natural way to describe, compose, and reason about the computation.

Synchronous dataflow

In single-rate synchronous dataflow, we assume that each transform operator takes in a single set of input tokens and produces a single set of output tokens. It is a simple generalization to allow the model to take in multiple tokens on a single stream link or to produce multiple tokens on an output stream link for one logical evaluation of the function. For example, a down-sample operator might read two inputs and only output one value, discarding every other input token. The number of inputs received from each input stream, or outputs produced on each output stream, can be different; for example, an operator might read two A tokens for every B token. However, as long as there are a constant number of

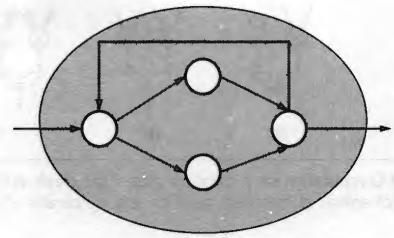


FIGURE 5.3 ■ A single-rate static dataflow graph.

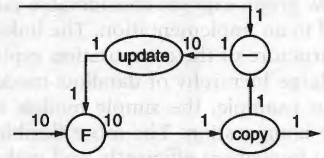


FIGURE 5.4 ■ A multirate dataflow graph.

tokens consumed from each input stream and tokens produced on each output stream on each such evaluation, the model retains the same power as before, but it now allows us to efficiently express multirate streaming applications; that is, some loops in the dataflow graph can operate at much lower frequency than others.

An inner loop might execute on every input to the graph, while an outer loop might perform updates only once every 10 inputs as shown in Figure 5.4. The numbers on the operator I/Os in Figure 5.4 indicate the rate of I/O consumption or production. The update module produces a single output every 10 tokens; the F function consumes a single input from update every tenth data input and output token; and the copy and subtract units each produce a single set of output tokens for each set of input tokens.

This is the Synchronous Dataflow (SDF) model [8], and it retains the same computational power of a finite automata. However, it allows multirate designs to be expressed more efficiently, explicitly identifying the relative operating rates of each of the computational functions in the graph. An implementation can use this information when provisioning operators and scheduling the sharing of physical resources. The computation is completely deterministic, and it is possible to automatically identify when operator rates are mismatched, leading to deadlock, and to automatically identify any buffering necessary during execution [9].

Dynamic streaming dataflow

Synchronous dataflow retains analysis simplicity because there is no data dependence in the consumption or production of tokens. Every evaluation of an object consumes and produces the same number of tokens regardless of the data.

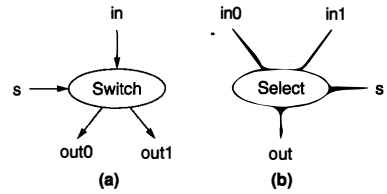


FIGURE 5.5 ■ The dynamic dataflow primitives—*switch* (a) and *select* (b).

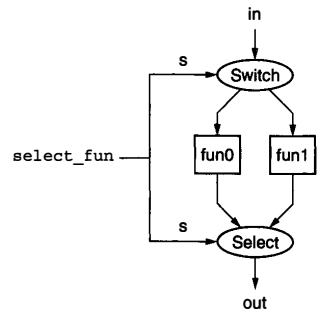


FIGURE 5.6 ■ Data-driven function selection in the dataflow model.

A more general model allows the production of input and output tokens to depend on the object state or the values of the inputs. We can fully capture this additional power by adding the *switch* and *select* operators, shown in Figure 5.5, to a persistent object graph.

In the figure, these two operators are data dependent, producing data on only one output, or consuming the inputs selectively, based on an input value. Equivalently, this can be captured by generalizing the notion of an object to allow its state to determine the token consumption and production actually performed on each evaluation. This allows us to efficiently deal with data-dependent cases, such as the following:

- Performing different operations based on the data (Figure 5.6).
- Varying the rate of the output relative to the input, such as in a compressor or decompresser (e.g., Huffman encoder).
- Iterating a computation a variable number of times to yield convergence (e.g., Newton–Raphson method for finding roots of equations).

In some cases these operations can be data independent, but only at the expense of more work (e.g., evaluating both functions in Figure 5.6 and then discarding

one result). However, if no constant bound can be placed on the iterations (e.g., the number of cycles required for convergence), data dependence is a necessity, not just an efficiency optimization.

The addition of data-dependent operators changes the power of the streaming dataflow, making it more difficult to analyze statically. The computation remains deterministic, but data-dependent production and consumption rates on operators necessitate reasoning about the streams as first-in-first-out (FIFO) token buffers. The addition of unbounded buffers between operators is sufficient to make the model Turing Complete, and it is no longer always possible to determine the FIFO buffers' required capacity. If the implementation buffer capacity is too small, the application may artificially deadlock. This demands either that the developer identify the necessary buffer size to avoid deadlock for each application or that the implementation provide dynamic support to allow arbitrary buffer expansion at runtime [10].

Dynamic Streaming Dataflow with Peeks

So far, we have demanded that the object evaluate based on a valid set of input tokens. In the data-dependent case, we allowed the value of the present tokens to determine which other tokens were consumed. We can further allow the operator to perform an action or modify state based on the absence of a token; that is, we can allow it to *peek* to see if an input is present. For example, a merge unit might have two inputs and forward either token to its output whenever there is some input present. As the merge unit example suggests, this creates new freedom for efficient evaluation but also introduces nondeterminism. The operator can now behave differently based on the arrival timing of its inputs. The data-dependent streaming dataflow model discussed earlier only introduced concern about deadlock but remained deterministic. The Dynamic Streaming Dataflow with Peeks model forces the developer to manage determinacy.

Streaming dataflow with allocation

The parallelism in the application is, in general, data dependent. Consequently, it can be useful for the operator dataflow graph to evolve on the basis of the data in a computation. In a telecommunications application, the number and type (e.g., voice, data) of connections change over time. Each channel has its own noise characteristics, perhaps requiring filter complexity (e.g., number of taps, length of echo cancellation) different from the others'. To accommodate these changes in the computational demand of an application over time, we must change the dataflow operator graph. We could force the graph construction to a different compute model and stay with graph evaluation as one of the models reviewed earlier. Alternately, we must expand the compute model with the ability to create new operators and link them into the graph.

The key addition now is for our operators to be able to perform instantiation (e.g., `new`) of operators and streams and to be able to connect them. Even if operators remain finite state, instantiation provides the ability to create unbounded state by growing the object graph to unbounded size, with arbitrary data structures implemented as subgraphs. This provides a more

efficient path to achieving Turing Completeness than the unbounded buffers in dynamic streaming dataflow.

While powerful, dynamic allocation means that the logical graph is changing during execution, and with dynamically changing computational graphs, it is no longer possible to optimize, schedule, place, and route them before execution. As a result, dynamic allocation gives us a model for the application to change during execution that can exploit the capabilities of a reconfigurable computing platform. However, it can also force a need for reconfiguration during execution, so allocation should be used with care. If it is infrequent, and allocated objects are long-lived, the cost of runtime management and reconfiguration can be amortized out over long usage periods.

General dataflow

Once we add allocation of operators, the model becomes powerful enough to be used as general dataflow computation. Some dataflow models do not treat operators or links as persistent (e.g., Arvind and Nikhil [11] and Culler et al. [12]). Rather, the dataflow is instantiated during a function or object call, used once, and then it is disposed. This does not change the model, but it does change the relative rate of allocation versus dataflow usage in a significant way. On typical reconfigurable platforms, dataflow construction is expensive, making it more difficult to efficiently map models that dispose of and reconstruct dataflow. For efficient execution on a reconfigurable platform, the compiler must discover opportunities to create dataflow operator graphs and reuse them across many invocations.

5.1.4 Sequential Control

The most widely used models for capturing and reasoning about algorithms are based on some form of sequential operation, including popular programming languages (e.g., C, Java, Fortran), control structures for hardware (finite-state machines), and formal models of computation (Deterministic Finite Automata, Sequential Turing Machines). The basic idea behind these models is that computations are defined as a sequence of primitive operations performed on some data state. The primitive operations define how state is transformed, including the state that determines which primitive operation(s) to execute next. Simple, concrete embodiments of this include sequential Instruction Set Architecture (ISA) processor models [13], but the state transforms can be much larger, may be coarse grained, and may include substantial parallelism on each sequential step.

Sequential control allows us to decompose a problem into simple, primitive operations. One thing happens at a time, making it relatively easy to reason about what each operation can do to the state.

Execution where only one primitive operation occurs at a time does not take full advantage of spatial reconfigurable architectures, leaving almost all the hardware idle as operations are sequentialized. Coarse-grained sequential operations that perform complex functions on large amounts of data may

provide sufficient parallelism to match the reconfigurable hardware. While strict sequentialization of operations defines the intended results in the model, careful analysis can often reconstruct a data dependence graph (Chapter 7), essentially the dataflow graph (see Section 5.1.3), to allow several operations to proceed in parallel and at the same time maintaining the sequential model semantics. Still, care must be taken in the sequential expression to avoid introducing false dependencies that inhibit parallelism. In general, the sequential expression can be a poor match for the parallel capabilities, and sequential models tend to lead the designer away from good reconfigurable implementations. There are, however, characteristics of our computations that sequential control may capture well at a high level.

- Data-dependent calculations are naturally captured with branching. Sequential control here allows us to express the selection of the computation we need to perform on the data.
- Phased computations where the algorithm does widely different things at different times may also be captured well with sequential control. If each phase requires widely different computation, spatially supporting them all at once may leave much of the reconfigurable hardware idle during the calculation. Transitions between phases gives us a way of expressing and identifying points in the program where it may be useful to reconfigure the hardware for the different portions of the task, instantiating only the relevant hardware for each phase.

Finite state
The simplest models of sequential control operate with a finite amount of state and are computationally equivalent to finite automata. Given this, verification of optimized computations can be performed in polynomial time with state reachability [14].

Sequential control with allocation
In more powerful models of sequential computation, we allow operations that allocate additional memory (e.g., `malloc`, `new`). Coupled with data-dependent branching, this allows the computation to allocate an unbounded amount of state, making the model Turing Complete, which in turn means that we cannot generally prove a bound on the amount of memory the application may require to run to completion.

Single memory pool
As noted earlier (see Section 5.1.2), because of an object's internal state we must carefully sequence the operations on it. We can think of each logical memory pool in a sequential model as an object with state so every operation on a single memory can be dependent on every other. If static analysis cannot prove that two users of the memory operator modify disjoint state in the memory, the operations must be sequentialized to preserve sequential correctness. In single-memory compute models, such as the C programming language or a traditional ISA execution environment, all memory operations must be sequentialized. This

sequentialization significantly limits the parallelism a compiler can extract from a single-thread, single-memory compute model. Consequently, large C programs that have not been carefully written to avoid these dependencies can be difficult or impossible to parallelize. Nonetheless, aggressive compilers can sometimes succeed in decomposing the monolithic memory into disjoint memory pools (e.g., Babb et al. [15]).

5.1.5 Data Parallel

Some applications are naturally captured as performing identical transformations on a set of independent data items. For example, we may need to perform the same color–space conversion to every pixel in an image, or perform the same match test to every data item in a database. Even though we could express such a task as a sequential loop over all the data items, it is often difficult for a compiler to prove the independence of each data item transform, and it can be tricky for the developer to identify which loop operations allow independent computation. Therefore, it is often useful to have an explicitly data parallel model that allows us to reason about and express algorithms as a sequence of transformations on aggregate datasets.

Once the desired computation is captured as a sequence of independent, identical, potentially parallelizable operations, we have considerable freedom in implementation for area–time tradeoffs. The computation can be rendered spatially and kept active as a heavily pipelined vector unit (see Vector coprocessors subsection of Section 5.2.4). Additional, parallel units can be allocated as the dataset demands and the platform permits.

The model typically remains sequential at the core and can suffer from artificial parallelism limits based on the provided sequential model. In particular, it may be hard to determine cases where multiple, independent data parallel operations can occur simultaneously. Although the parallelism on a single operation is limited by the size of the aggregate data item, the data parallel model does give general high-level guidance to the developer that often trends in the right direction for efficient spatial realizations.

5.1.6 Data-centric

In the streaming dataflow model, the designer thinks of the application as a transformation graph with data generally flowing through operators with fixed state. For some applications, such as physical simulations, it makes sense to turn that around and think about the operators and their state as the primary data structure, and reason about the computation as transformations on the operator state. For a network flow problem, we might construct the graph for the network; each operator maintains state to represent the flow through its links and the accumulating overflow at the node, and each operator sends tokens over the edges between operators to reroute flow. At each sequential step we may allow each operator to process a set of inputs and send a set of outputs. At a high level the operation is data parallel, with each operator performing its node update operation; however, locally the computation may be data and state

dependent. High-level data parallel instructions to the operators can sequence phases of the computation (e.g., preflow and push phases in network flow).

Applications that regularly visit many nodes on large graphs of data are a natural source of parallelism. Even if the nodes are not identical, there are usually only a small number of different node types, providing an opportunity for sharing of spatial operators. Without strict dataflow communication ordering, additional disciplines may be necessary to maintain determinacy. Efficient execution may require load balancing and sharing if graph nodes have low or widely varying activity factors.

5.1.7 Multi-threaded

A widely used model for parallelism is multi-threading or some form of CSP [2]. Basically the model is a collection of sequential control processes with communication links between them, either as direct communication edges or as shared memory. Multi-threading is a very general model and, in fact, any of the models presented so far could be seen as subsets of it.

The problem with multi-threading is that it is too general and powerful to provide guidance for application development and correct implementation. It permits the expression of solutions that are difficult to reason about, and it provides little guidance on good solutions and guaranteeing determinism [4]. How should the application be divided into threads? How do the threads synchronize with each other? How do we guarantee determinism and avoid deadlock? In our streaming dataflow model, we think of each thread, the operators, as transforms on the data flowing through them, and we synchronize based on token flow; in our data parallel model, we think of each thread as a separate data item and update each in lockstep; in our data-centric model, we think of each thread as an active object in the graph, performing updates on barrier-synchronized steps.

When faced with applications that demand more power than is available in a more restricted model, we should think about the power actually necessary for our application and the extent to which we can define a restricted discipline for using the multi-threaded model that answers the questions the model does not answer for us. What do our threads and operators represent? What is the synchronization discipline? What is our basis for reasoning about determinacy, deadlock, and correctness?

5.1.8 Other Compute Models

The compute models reviewed here are by no means exhaustive. From the start, we want to emphasize the need to consider multiple models and choose the one most natural for the application. The set just described are useful in reasoning about the architectures and applications developed in this book and may be most helpful for reasoning about reconfigurable applications. Nonetheless, as we master these models and encounter applications that match poorly with them, we should look for others that further ease the conceptualization of an

application. (For other summaries of compute models see Lee and Sangiovanni-Vincentelli [16] and Lee and Neuendorffer [17].)

5.2 SYSTEM ARCHITECTURES

Whereas the compute model helped us understand the natural composition and parallelism in the application, the system architecture deals primarily with how we organize the implementation. As noted (introduction to Section 5.1), applications in a compute model may be mapped to any of several system architectures. The choice of architecture will depend on technology costs and resource availability compared to the application resource and performance requirements. For example, a platform that is very small compared to the size of the task drives serialization in the implementation, which may favor sequential control. Even here, though, we have important decisions to make about the level at which the sequential control is exercised (e.g., coarse-grained phasing) (see Phased reconfiguration manager subsection of Section 5.2.2) versus cycle-by-cycle sequencing (see FSMD, VLIW datapath control, and Processor subsections of Section 5.2.2).

Figure 5.7 is an overview of the system architectures, and their variants, covered in this section. To help the designer easily identify those that may be relevant to his or her specific problem, we open the description of each one by identifying the major problem or challenge it addresses.

5.2.1 Streaming Dataflow

We best exploit a reconfigurable platform when we can spatially arrange specialized computational pipelines and keep them each actively working on useful computation at a high cycle rate. How do we organize computations that can exploit this efficient use and arrange for data to feed the pipelines?

In the simplest case, we can use one of the streaming dataflow compute models (Section 5.1.3) directly as a guide for system implementation; that is,

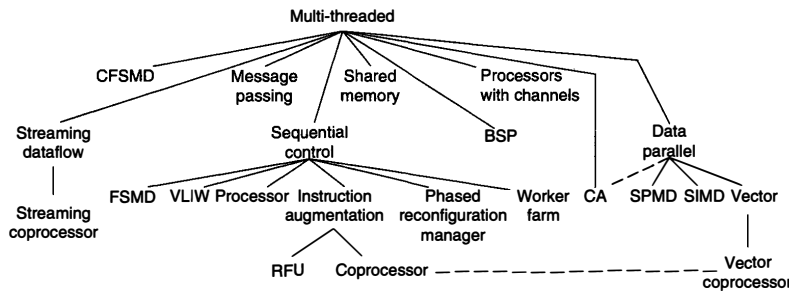


FIGURE 5.7 ■ Overview of system architectures.

we can map each operator to its own physical datapath and interconnect them all via configured interconnect. The efficiency of spatial pipelines on FPGAs and reconfigurable architectures makes this attractive. Further, the streaming model shows where in the detailed, cycle-by-cycle behavior of operations we have the implementation freedom to adapt to target platform delays. This architecture is known as *Pipe and Filter* in the literature [1]. Chapters 8 and 9 describe applications and programming that use it.

In the remainder of this section, we highlight four detailed techniques that are often useful in implementing streaming dataflow architectures.

Data presence

Direct connections of pipelined datapaths may pose challenges to guaranteeing the proper streaming dataflow semantics, offering efficient implementations, or allowing composition. These challenges include:

- Configured interconnect paths between operator datapaths may be long and can vary on the basis of platform, implementation technology, and operator placement. Long interconnect paths may limit the speed of operation.
- Different implementations of an operator may operate at different rates, and we want to be able to interchange these implementations without redesigning the implementations for all of the operators that interact with this operator.
- In dynamic dataflow models, an operator may not be able to consume an input, or produce an output, on every cycle of operation.

To promote easy and efficient operator composition, we can associate a "data present" signal with each data item. We design the physical functional units so that they can stall while waiting for the required inputs to be present. This decouples the clock cycle for interconnect and compute from the logical alignment of data, allowing us to pipeline the datapaths and the interconnect paths between them without changing the meaning of our computation. In many cases, we need to treat the interconnect paths as FIFO queues between operators; further, we can use back-pressure to indicate when a stream link between operators is full and so the upstream operator must wait before producing additional results.

The discipline makes the implementation of an operator independent of the implementation of others with which it communicates, allowing each to run at its desired clock rate even as all of them are composed together to build a larger system. This permits a variety of composite implementations:

- Operators and interconnect can all be designed to a single target clock frequency.
- Operators may run on separate clocks that are based on a common base frequency.
- Operators and interconnect may run fully asynchronously, handshaking locally.

- Operators may use a Globally Asynchronous, Locally Synchronous (GALS) model, with local operator clocks and asynchronous handshaking between operators.

It is still necessary to pay attention to the length of logical cycles in the original streaming dataflow graph; a loop in the graph may force sequential evaluation of all the graph's operators. Even though we can physically pipeline the operators and the links, the logical alignment of data may force the operators to effectively operate at lower rates, leaving the datapaths and interconnect inactive on most cycles. Such dependencies may motivate sequential sharing of operators or the resources inside them.

Datapath sharing

Ultimately, we must fit our entire dataflow graph onto our physical platform. For efficiency, we hope all of the hardware allocated to the dataflow graph is put to productive use on each cycle. Following are specific scenarios we may need to address:

- The substrate may not be large enough to hold the entire dataflow graph spatially.
- Multirate dataflow graphs may leave some operators idle while others are busy.
- Cyclic dependencies in the dataflow graph may make it impossible to keep all the operators active simultaneously.

To use the datapath hardware efficiently in cases such as these, it is often useful to share a physical datapath among multiple operators. In the simplest case, we share identical operators so that the datapath remains the same, only adding the unique state associated with each of the operators. In more complicated cases, we might generalize the datapath so that it can implement two or more types of operators.

When we share operators, we need to identify which data inputs are associated with which logical operator. This can be simply orchestrated by scheduling and pipelining for static-rate operators, but for dynamic operators and variable implementation delays, it may be necessary to further tag the data with information that identifies the logical operator for which it is destined.

Streaming coprocessors

With extreme variation in operator frequencies, large numbers of operators, and very small platforms, operator sharing may not be sufficient to provide an efficient solution. Here, even allocating a single datapath for a particular hardware type may leave the datapath highly underutilized or it may still demand more area than the platform provides.

In these more extreme cases, it is often useful to schedule the low-rate operators onto an embedded or attached processor (see Processor subsection of Section 5.2.2). By augmenting the processor with streaming instructions, processor-mapped operators can communicate efficiently with streaming

dataflow. Data destined for active operators can be forwarded spatially, while data intended for inactive operators can be queued in memory. Data presence allows the processor tasks to operate without knowing the size of the reconfigurable platform or the residency of operators. Data presence on stream reads by the processor can be used like a memory stall, tolerating varying implementation delay on the reconfigurable platform or triggering an operator swap, similar to a thread swap on an I/O or virtual memory page miss.

Interconnect sharing
In spatial computations, interconnect often consumes a substantial portion of the hardware area and can often be a performance bottleneck. Consequently, we should always be concerned about using the interconnect efficiently. A direct, configured connection between a source and a sink can be inefficient when

- The link between operators is used infrequently because of a slow datapath or a low-rate operator relative to the rest of the computation.
- Because of dynamic data dependence, the communication rate on many links is highly variable.

To optimize interconnect in these cases it may be possible to reduce the interconnect requirements on these interconnect links by sharing them. Links can be shared in a variety of ways, including shared bus, pipelined ring, and network-on-a-chip. These can be statically scheduled in data-independent cases and in data-dependent cases with low communication variability, or dynamically managed when the data-dependence produces high variability.

5.2.2 Sequential Control

While sequential control is familiar and heavily used for highly sequential machines and algorithms, it is most interesting to us as a way to organize synchronization and control of a large set of spatially parallel operators, particularly when

- The compute task is too large to fit spatially onto the available computing resources, so we must share the resources in time.
- Data dependencies result in low utilization of the datapath, so we can share resources to produce a smaller design with little or no impact on compute time.

Even when we start with a dataflow or data-centric computation, it may be useful to control the implementation, or parts of it, in sequential manner; this is especially true when we share spatial operators in time to economize on space.

A common idiom is to

1. Start with the computation data dependence graph (e.g., Figure 5.8 (a) or Figure 5.2) based on the description in the compute model.
2. Identify a base set of datapath elements that can implement all the operators in the computation graph.

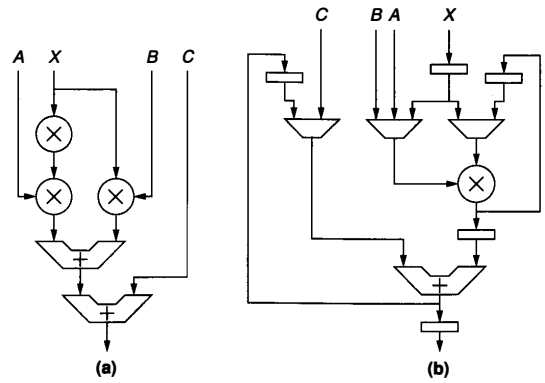


FIGURE 5.8 ■ A dataflow graph for $y = Ax^2 + Bx + C$ with three multiplies and two adds (a); a shared datapath (b) with a single multiplier and adder with state registers and multiplexors.

3. Schedule the operators in the compute graph onto the datapath elements.
4. Add data storage and interconnect to hold intermediate operator state and forward data between the locations where producing and consuming operations are performed.

In the simplest case, we might allocate a single datapath element for every operator in the compute graph. While there is no sharing in this case, it may still be necessary to control when the elements should sample their inputs and produce outputs. This can be done in a purely dataflow manner as suggested in the Data presence subsection of Section 5.2.1; however, for modest blocks in a single clocking domain with predictable datapath timing, it can be more efficient to centrally control the operators, sending control signals to each datapath element from a central control unit.

In the more general case, we have fewer datapath elements than operators and must orchestrate the sharing of those elements and interconnect. Intermediate values in the original computational graph that are not consumed in the cycle immediately following production, or immediately after being routed from the source to the destination, are stored temporarily in memories (see Figure 5.8). Object state that persists through the computation must be stored in memory or registers and routed to the associated datapath when the operator has its turn to use the datapath.

Within this paradigm, the key piece of freedom is the selection of the base datapath elements and the assignment of operators to them. This selection is where we can exploit area–time tradeoffs, allocating more spatial datapath elements as we have more area available and want to reduce the

time for computation; it is also where we have opportunities to instantiate highly specialized operators that are matched to the needs of a particular task (e.g., Chapter 22).

The design community has identified a number of stylized forms for sequential control over the years. In the remainder of this section, we highlight a number of organizations and note when they may be useful for managing reconfigurable resources.

FSMD

Once we have selected the operators, assigned them to datapath elements, and scheduled the operations, we still need some way to implement the central control that manages resource sharing and orchestrates the routing of intermediate data among datapath elements.

One common way to support this control is to build a finite-state machine (FSM) that controls the operation of the datapath; this is called a Finite-State Machine with Datapath (FSMD) [18]. The FSM controller can assert the various controls (e.g., multiplexer selections, load or read/write enables, datapath operation selection) on each cycle and provide cycle-by-cycle sequencing of them (see Figure 5.9). Further, the FSM can take inputs from the datapath and, based on their data, branch to different control sequences.

A data-dependent operator might be internally implemented as an FSMD, with the state transitions in the FSM controlling the input consumption and output production (see Dynamic streaming dataflow subsection of Section 5.1.3, or Section 5.1.6).

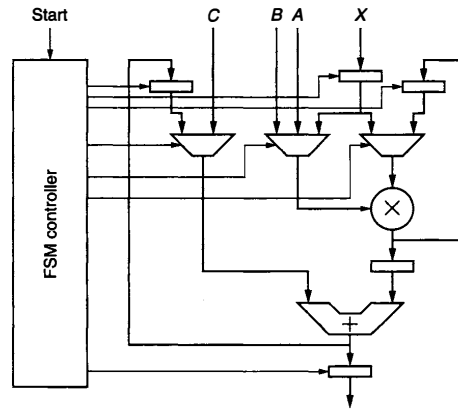


FIGURE 5.9 ■ FSMD for a single multiply and add datapath for quadratic equation evaluation.

VLIW datapath control

While we can build a custom FSMD for each application, the FSMD form does not, itself, provide disciplined organizations for state storage and data routing, nor does it suggest any organizing principles for managing the control of each datapath. As a result:

- With heavy sharing there is a proliferation of intermediate state that needs to be managed.
- With many datapath operators, state memories, and switched interconnect, there is a proliferation of control signals that must be distributed to these compute, memory, and interconnect elements.
- For generality, robustness to change, and the opportunity to deploy the datapath for multiple tasks, it may be useful to be able to change the control sequencing without rebuilding the entire controller.

One stylization for sequential control is the Very Long Instruction Word (VLIW) model, which in its most primitive form is closely related to Horizontal Microcode [19]. In VLIW we start with the collection of datapath elements as before. These can be homogeneous or heterogeneous and provisioned according to the needs of the task. We then add one or more memory banks to hold inputs to each datapath element, and we add switched interconnect between the datapath elements and the memories. The controls to the memories, datapath elements, and interconnect become the long instruction word, to which we allocate a wide memory, perhaps distributing it with the memory cells and memory outputs local to the compute, interconnect, and memory elements they control (see Figure 5.10). To issue an "instruction" (see Chapter 36), the controller sends a single instruction address to the wide memory, and the memory output tells every datapath element, memory, and interconnect switch how it

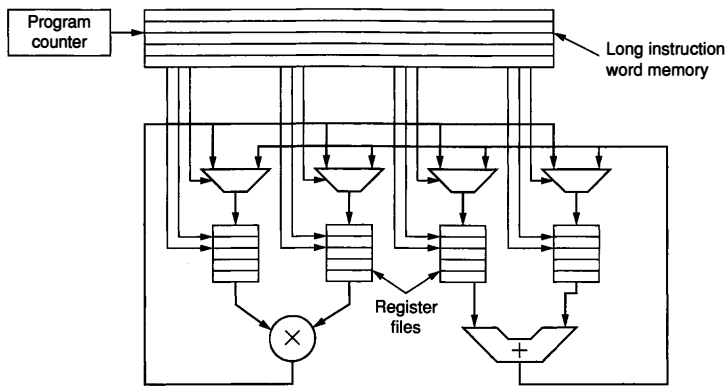


FIGURE 5.10 ■ VLIW-style control of a single multiply and add datapath.

should be configured on that cycle of operation. Typically, the datapath is configured to send one or a few bits back to the controller that can be used to select the next instruction address to allow data-dependent branching.

When VLIW was first introduced for general-purpose processing (e.g., Ellis [20]), the datapath elements used were generic (e.g., ALUs, FPUs, load/store units), of modest size, and fairly homogeneous. With FPGAs and reconfigurable architectures, we have the opportunity to select the datapath elements based on the task, make them highly specialized, and potentially even make them fairly coarse grained (e.g., a DCT step, motion estimation step, or AES encryption step).

Processor

The FSMD and, to some extent, VLIW control both assume that there are common datapaths that can be shared, and both allow multiple, concurrent operations to exploit the spatial parallelism available on an FPGA or reconfigurable device. However, for some computations our premium may be space saving rather than operation performance. That is, overall system performance may depend on this operator fitting onto the platform and being performed infrequently, but the time the operator takes may have little impact on it.

A conventional, sequential processor or microcontroller with a single arithmetic logic unit (ALU) is the extreme end of sharing, where we

1. Allocate a single, universal datapath element.
2. Decompose all operators into sequences of operations on this primitive datapath element.
3. Provide state storage for all intermediates between the cycle of production and the cycle of consumption, including storage for all object state.
4. Define a narrow instruction to control the datapath element and state storage.
5. Provide a sequencer and branch unit to sequence the instructions on the datapath in a potentially data-dependent manner.

Because this allocates minimal area to computation and interconnect, the total area for the computation can be very compact; however, compactness comes at the expense of most of the resources going to control, instruction, and state management. As a result, only a tiny fraction of the consumed computational resources go directly to implementing the application (see Chapter 36).

If heavy serialization to economize area is what we need for an entire task, a dedicated processor is certainly more efficient than a processor configured on top of an FPGA. Nonetheless, there are a few scenarios where a processor configured on top of an FPGA might be reasonable. Such scenarios would typically exploit the flexibility of either building a particularly specialized and lightweight processor for a specific task and/or embedding one in a flexible and highly integrated manner alongside a much larger computation implemented using a more spatial implementation architecture.

When we have multirate computations (e.g., Synchronous Dataflow model subsection of Section 5.1.3), some operators may execute at much lower rates than others. To balance the system and achieve maximum application performance in a limited area, we typically allocate space to operators in proportion to the fraction of the total computation they perform. As a result, we may end up with some very infrequent operators that are needed to complete the task but can afford to operate very slowly. If there is a dedicated, attached processor, perhaps these operators can be run there; if not, or if the flexibility to place the processor datapath for this operator local to other computations is important, it may be worthwhile to implement the operator as a configured processor.

Instruction augmentation
For resource sharing, a sequential controller is often necessary to direct the use of specialized datapaths. Sometimes this takes the form of a mix of irregular, low-throughput tasks that do not need to be executed quickly along with some very regular computations that are critical to performance. Manifestations of this need include:

- We need to sequence a modest amount of FPGA or reconfigurable logic.
- The computation contains a few operations that account for most of the time, embedded in a large amount of irregular tasks necessary to define the complete computation.

A processor is an efficient, programmable, and well-understood sequential controller. Consequently, it is often useful as the base design for a sequential controller. This is common enough that many platforms provide a dedicated processor attached to an FPGA or reconfigurable array (Chapter 2). It is also useful enough that this may be one of the motivations to employ a custom, configured processor.

One way to provide the coupling between the processor and the FPGA array is to treat the functions provided by the FPGA as additional instructions that augment the processor's base instructions. The processor's execution model of issuing instructions and expecting them to be performed in sequence remains intact, but the set of instructions it can issue are enlarged by the configured array. The FPGA instructions can potentially be very powerful, performing the equivalent of hundreds of base processor instructions in a single invocation. This can be particularly effective when a few such powerful instructions can cover the bulk of the execution time in the task. The processor serves as the application glue, sequencing these dominant operations and orchestrating the movement of data to connect them.

Functional Unit model One way to implement instruction augmentation is to provide a reconfigurable functional unit (RFU) (e.g., Razdan and Smith [21], Hauck et al. [22], and the Tightly coupled RPF and processor subsection of Section 2.2.2); that is, we treat the reconfigurable array just like any other functional unit

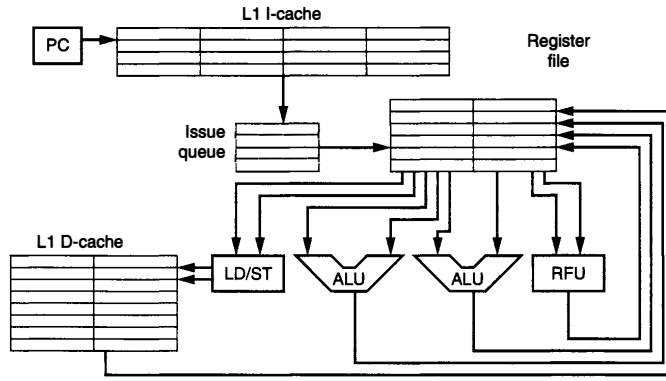


FIGURE 5.11 ■ A super-scalar processor with an RFU.

in the processor (see Figures 2.12 and 5.11). The processor issues instructions to it, feeding it data from the register file, and the array returns the result to a register. Normal processor issue and scoreboarding mechanisms can be used to accommodate variable delay in the array operation. The Functional Unit model may be particularly useful in specializing a configured processor to a particular application, where the custom functional units each perform a single function. It can also be used for coupling a custom processor to a reconfigurable array. One variant is to allocate a set of opcodes in the instruction for the reconfigurable function unit so that the processor instruction can call out different array operations.

The Functional Unit model is easily integrated into a conventional processor pipeline. However, it provides limited I/O between the processor and the array and demands that the reconfigurable operation be a function, preserving no internal state. This potentially limits the use of the array, by preventing the allocation of large, coarse-grained operations on it.

Coprocessor model Another way to implement instruction augmentation is to treat the reconfigurable array as a coprocessor (e.g., Callahan et al. [23]—see Figure 5.12), with the processor performing explicit data moves to and from it and directing it to perform specific operations. The coprocessor model allows the array to hold its state and places data close to it. This makes it possible to push larger portions of the computation onto the array, only communicating data back to the processor at large operation boundaries. The I/O to a single operation can be sequenced over several cycles, which allows greater flexibility in operator granularity.

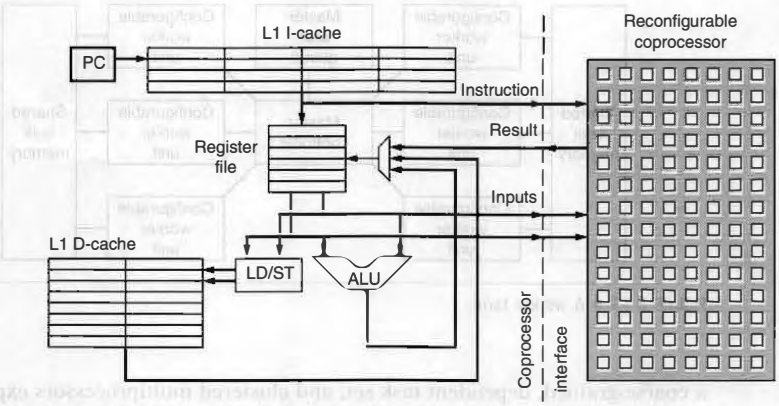


FIGURE 5.12 ■ A scalar processor with a reconfigurable coprocessor.

Phased reconfiguration manager

In the preceding sections, we shared the FPGA or the reconfigurable array resources in time in a fine-grained manner by scheduling operators on a cycle-by-cycle basis onto the datapath elements. This works when we have common operator types that permit sharing, or when we can generalize the datapath element to support many operators. In order to realize this we added additional circuitry to the design to flexibly route data between the datapath elements and to sequence the sharing. These additional resources did not contribute computation to the original task and so were pure overhead. However, since our hardware is reconfigurable, in some cases it is possible to reconfigure it and perform this sharing at a coarser granularity with less overhead. Since reconfiguration is often slow, this is viable only when we can arrange for the array to be used for a long period of time in a single configuration, such as when tasks operate in phases, performing distinct computations for long times. For this to be useful the "long time" in a configuration should be long compared to the time required to perform the reconfiguration (see Section 4.2 and Chapter 9).

In these cases, sequentialization is very coarse grained. We can nonetheless still think of the sequencing as a sequential control application, with each state potentially representing a different configuration of the array. The sequential controller monitors the execution to detect the end of the phase, implements configuration, and may even perform state-dependent branching. Sequential control can be realized with many of the architectures previously discussed (e.g., FSM, processor, instruction augmentation).

Worker farm

Sometimes we may have a set of dependent operations where each one runs for a large and variable amount of time. For example, Unix/Linux `make` rules specify

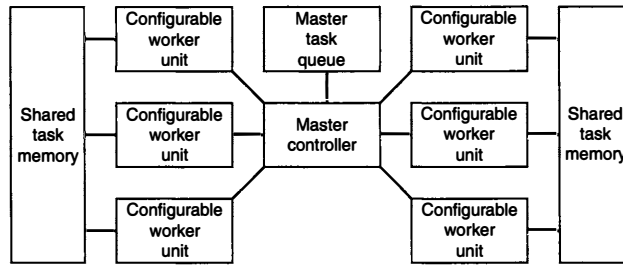


FIGURE 5.13 ■ A worker farm.

a coarse-grained, dependent task set, and clustered multiprocessors exploit this parallelism with parallel `make` utilities such as `pmake` [24]. The variable runtime means that predetermined assignments of operations to hardware resources can be very inefficient.

We can exploit the reconfigurable hardware in these cases by organizing resources as a set of *workers*, which actually process jobs, and a central *manager*, which is responsible for assigning operations to them, potentially coordinating data movement and reconfiguration (see Figure 5.13). Here, the manager might

1. Maintain a queue of ready tasks.
2. Issue the first ready task to execute on a free worker.
3. Continue issuing tasks to workers until there are no free workers.
4. Wait for one or more workers to signal completion.
5. As tasks complete, put any tasks they enable on the ready queue.
6. Loop back to step 2.

Operations are enabled in dataflow form as they are completed. If the tasks are largely homogeneous or taken from a small set of types, the workers may be identical or taken from a small set of datapath configurations. If they are long running and highly heterogeneous, it may make sense to reconfigure them to each task; when the reconfigurable array supports it, this might include partial reconfiguration (see Section 4.2.3) of the array to customize each worker for its next task.

5.2.3 Bulk Synchronous Parallelism

In our sequential control architectures, we had a central controller telling every datapath element what to do. This guaranteed that the datapath elements moved forward in a synchronized manner. However, if the work required by each datapath is highly data dependent, a centralized locus of control may become inefficient. Consequently, we often allow the local datapathsto have independent control but

still want to guarantee that they remain synchronized at some coarser granularity. In particular, we might want to ensure that one set of tasks completes before another begins.

Bulk Synchronous Parallelism (BSP) [25] can be seen as a variant that keeps the synchronization centralized but distributes the datapath sequencing. In BSP, independent units of computation progress independently, with the local computations punctuated by periodic barrier synchronization events. Each local computation announces when it reaches the barrier and waits for a global acknowledgment that all local tasks have reached it before proceeding.

The barrier is an efficient technique for supporting data-dependent, time-variable operations in each task while still providing strong synchronization guarantees. An alternate would be to statically determine the length of each epoch and have local tasks that complete their epoch early wait until the static epoch duration completes. If the runtime of each task varies widely based on data or potential resource contention, the static bound necessary to guarantee correctness may be excessively long compared to the common case local task completion time.

Further, if the local tasks are Turing Complete, it may not be possible to even identify such a static upper bound on the timing between barriers. The expense of the barrier is that it requires $\Omega(\log(N))$ time to perform the synchronization in the ideal case, where wire delays are negligible, and $O(\sqrt{N})$ or $O(\sqrt[3]{N})$ time in realistic 2-space or 3-space physical implementations for the barrier to complete. This suggests efficient operation only when the computational work between barriers is at least as large as this barrier synchronization time.

A BSP architecture can be appropriate for implementing data-centric computations (Section 5.1.6). Often objects communicate over their connected graph links. For many applications it is useful to guarantee that each object processes one round of method invocations before starting the next round. Barriers between rounds allow the operator to know when it has received all the invocations associated with a single round and can safely advance [26].

5.2.4 Data Parallel

As the Data Parallel Compute model suggests, sometimes computation can be organized as a set of computations applied, mostly independently, to a large set of data (see Section 5.1.5). This gives us both parallelism and regularity that a reconfigurable implementation can exploit. We want to be able to use this parallelism in a scalable manner, allocating more or less hardware as the platform permits.

A number of stylized architectures support data parallel computations and can be tuned for varying amounts of parallelism. The remainder of this section highlights three architectures and one technique for interfacing and controlling data parallel computation with more general computation.

Single program, multiple data

Although it is sometimes useful to apply the same basic operations to each component piece of data, these operations can be highly data dependent and can benefit from independent, local control. However, even though they are locally independent, it may be useful to guarantee that a set of operations on the data completes before continuing with the next operation set.

SPMD (single program, multiple data) is an organizational structure that follows the high-level Data Parallel model with minimum stylization within each data parallel task. Essentially, we have a collection of independent threads or control units that happen to be performing the same operation on different datasets. Individual independent threads can, themselves, be implemented as one of the system architectures described here. They are typically synchronized periodically in BSP fashion (Section 5.2.3).

Single-instruction multiple data

Control and instructions for a datapath can become expensive. Thus, if the data dependence for data parallel operations can be kept low, it is beneficial to share instructions and control across a large set of datapaths.

SIMD (single-instruction multiple data) architectures control the hardware operations on a cycle-by-cycle basis similarly to our sequential control architectures (Section 5.2.2). However, instead of a heterogeneous set of datapath elements, each potentially receiving unique operations, a single, common instruction is delivered to all of them. Each element has its own data and performs the sequence of instructions on it. Communication between datapath elements is also supported with common instructions to orchestrate data movement.

SIMD architectures can be more compact per processing element than VLIW architectures, because they do not need to store separate instructions for each compute, memory, or interconnect block. However, since SIMD architectures force all datapath elements to perform the same operation simultaneously, the SIMD datapath elements are efficiently utilized only on much more stylized and limited computations (see Chapter 36).

Chapter 10 describes a particular SIMD system in more detail, including an approach to SIMD compilation for FPGAs.

Vector

The motivation for vector architectures is similar to that for SIMD: When operations are sufficiently regular and data independent, they admit implementations that economize on resources by sharing instructions and associated control. Vector architectures particularly exploit the fact that datapath operations often have long latencies and can be pipelined so that calculations on many, independent data items can reuse the datapath at high throughput.

In a vector organization, a sequential controller issues data parallel instructions across a logical dataset. Here, we think of supplying vectors of component data, rather than individual words, as our inputs and outputs of instructions. The instructions perform operations similarly to a sequential processor on the pairwise components of vector inputs. Rather than the data living with the

datapath elements, as is typical in SIMD, the vector data is normally kept in central memory banks and vector register files and is routed to the datapath elements. The vector instructions then specify where to find vector inputs and where to return vector results. The data parallel operation on these vectors can be performed in sequence on a highly pipelined vector functional unit, in parallel on a set of parallel functional units, or as a sequentialized set of parallel batches based on the area allocated.

On reconfigurable platforms, we can construct highly specialized vector functional units for each task. Thus, a vector control unit can be augmented with specialized vector pipelines just as a processor can be augmented with configurable instructions in an Instruction Augmentation architecture (see Instruction augmentation subsection of Section 5.2.2). Here we are operating on vectors of data rather than on individual scalar data elements. As with other models, we can identify the coarse-grained, data parallel operations required in the task and allocate a suitable set of functional units for them. The vector control unit then issues instructions to perform the data routing and sequencing to connect the operations running on the vector functional units (see Figure 5.14).

Vector coprocessors

As noted earlier, we often have a mix of irregular computations and more regular stylized computations (see Instruction augmentation subsection of Section 5.2.2). This is certainly true when exploiting highly stylized, data parallel computations using vector or SIMD architectures.

The Coprocessor model (see Coprocessor model subsection of Section 5.2.2) provides one stylized way to add configurable vector units to a base processor architecture (e.g., Wawrzynek et al. [27] and Jacob and Chow [28]). Here, the vector operations become coprocessor instructions. The processor can remain scalar, with normal instructions and register files, with the configurable vector unit maintaining all the vector states local to the configurable array. The vector

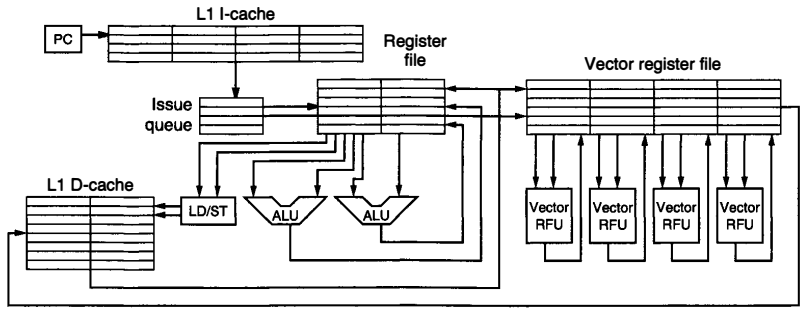


FIGURE 5.14 ■ A super-scalar processor with vector functional units.

coprocessor can keep multiple vector operations in flight, using scoreboarding on memory or vector registers to enforce sequential semantics on the sequentially issued vector operations.

5.2.5 Cellular Automata

Although spatial computation organizations offer great parallelism, they also demand that the spatially distributed datapaths communicate with each other. For large computations, the physical latency between distant operations can be large; further, the worst-case, cross-chip latencies actually grow relative to cycle rates as technology scales. Considerable, nonlocal traffic can slow the computation both because of round-trip latencies and because of limited available cross-chip bandwidth.

Cellular automata (CA) suggest a pattern for organizing computations as a line (one dimension), mesh (two dimensions), or cube (three dimensions) of regular operators with nearest-neighbor communication (see Figure 5.15). The operators run logically in lockstep, sampling the state of adjacent operators and updating their own. The regularity of identical operators makes it easy to scale to larger spatial designs. Moreover, nearest-neighbor communication eases layout and guarantees that communication does not limit overall design performance. A CA can be seen as a very stylized data-centric (see Section 5.1.6) computation in which the parallel operators have a restricted, regular communication pattern.

The restriction for nearest-neighbor communication may seem extreme, but it naturally shows up in many physical world simulations. Because physical interactions are also primarily nearest neighbor, the topology of the physical problem often maps directly to that of a regular CA. Examples of physical simulations include discrete-time solutions to wave, diffusion, Navier–Stokes, or Maxwell's equations (see Chapter 32). Perhaps the simplest and most well-known CA is Conway's game of "Life" [29]. It is even possible to implement CAD optimizations,

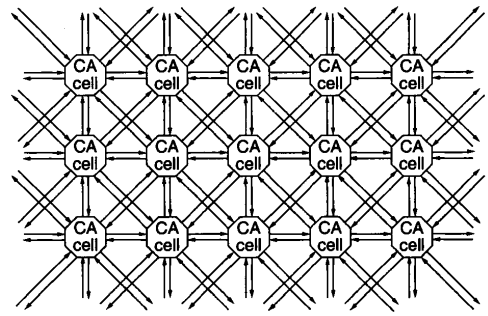


FIGURE 5.15 ■ Two-dimensional cellular automata.

such as placement, using CAs (e.g., Wrighton and DeHon [30]; also mentioned briefly in Chapters 9 and 20).

Folded CA

CAs can be highly efficient, but the size of the fully spatial design depends on the size of the problem. Thus, for large problem sizes the fully spatial CA can be too large for an affordable reconfigurable platform.

Since the CA is based on an array of identical operators and regular communication, it can be efficiently folded onto smaller physical platforms (e.g., Margolus [31] and Kobori et al. [32]). At one virtualization extreme we can build a single, physical CA cell processor and stream through the state of the virtual cells in series, using the single physical cell to implement all cells. The access pattern for the data is regular and predetermined, allowing efficient use of memory bandwidth. Plus, all the data communications are local, which means that we can readily program data buffering so that all data is available at the cell as needed on a single pass through the memory. We can also implement a single row (or column) of the CA and scan through one row (or column) at a time. In fact, we can choose just about any number of physical cells to implement—up to half the number in the logical array—and achieve a linear speedup of the computation. Chapter 32 describes a particular, folded mapping for a finite-difference, time-domain solution of Maxwell's equations.

5.2.6 Multi-threaded

The architectures discussed previously all place restrictions and stylizations on the computation to allow efficient implementation. Nonetheless, particular restrictions may not match with the needs of some applications and, consequently, may not provide the most efficient implementation support.

As suggested with compute models, multi-threading can be seen as the most general organization (Section 5.1.7). It provides great expressiveness, but at the cost of little guidance to the designer in how to exploit that expressiveness and guarantee correctness of implementation. This expressiveness can also make it expensive to support the full generality of the model on reconfigurable hardware.

In the remainder of this section, we review some common multi-threaded organizations and the benefits and caveats they entail.

Communicating FSMs with datapaths

Earlier we noted that an FSMD is a stylized way to control the operation of a datapath (see FSMD subsection of Section 5.2.2). For very large designs, a central controller may become a performance bottleneck for the following reasons:

- Central control may lead to unnecessary state explosion in a central controller.
- Sending control signals across a large system to a central controller and distributing control back from it may result in long latencies and slow operating rates.

One alternative to a single controller is decomposing the system into a number of independent FSMDs that communicate with each other. In this way, in addition to its own datapath controls, the FSM controller for each FSMD now contains inputs and outputs to one or more of the other FSMDs through which it coordinates synchronization. Thus, each FSM controller can be simpler and faster than the single, monolithic controller, and each can branch independently. However, the designer must be careful to manage the coordination of the FSMs so that they do not deadlock or otherwise transition into inconsistent states.

Technically, a composition of finite automata is still a finite automata, and it is possible to compute the composite automata in order to prove properties of the composite system. The state space of the composed automata can be as large as the product set of the state space of the individual automata. In some cases this state explosion can become intractably large for practical verification.

Processors with channels

In the Processor subsection of Section 5.2.2, we saw the motivation for an operator or several operators to run on a processor or, more likely, a processor controller with a specialized datapath. For similar reasons that motivate the communicating finite-state machines with datapaths (CFSMD) described in the previous section, it may not make sense to centrally control this collection of processors.

Here, too, we decompose the computation into a collection of augmented processor datapaths that coordinate with each other through direct links. Special instructions allow the processor to poll information from input channels and place information on output channels.

Message passing

When we connect processors or FSMs with communication channels, we often find it inefficient to commit dedicated, point-to-point links.

- The data rate on point-to-point channels between processors, operators, or FSMDs can often be too low to merit a dedicated channel.
- Dedicating point-to-point channels between processors, operators, or FSMDs can be too expensive for an implementation.
- Individual units of control may only need to communicate infrequently.

Rather than keep a channel open at all times, operators can share a common communication infrastructure (e.g., bus, pipelined ring, network-on-a-chip) and send their coordination information tagged with the identity or location of the recipient—in other words, send messages.

Shared memory

Multiple operators cooperating on a task may need infrequent access to a large set of shared state. When we exploit parallelism, these operators may be running on different parts of a physical platform yet need to access shared data pools.

When possible, it is best to give ownership of state to a single operator and let it provide coordinated access to it. This approach avoids a host of synchronization

problems that can make parallel execution particularly troublesome. If the state is small and infrequently changed, and when several operators need regular access to it, it can make sense to allow each to have its own copy and allow coordination operations to change the data across them. However, when the state is large and infrequently accessed, such as with a large database, it is sometimes efficient to allow multiple physical processing elements to access a single, shared memory pool to avoid replicating the data and to allow data communication to be deferred until it is needed. This can allow each processing element to extract just the information it needs without burdening the others with knowing which information will be needed by which processing element.

In the general case, we may share memory pools between small sets of physical processing elements. Unlike with homogeneous multiprocessors, there is generally little reason to have a single, large, shared memory pool across all the processing elements in a reconfigurable computer. The configurability of our reconfigurable designs allows us to limit sharing based on the shape of communications in the application.

5.2.7 Hierarchical Composition

In this chapter we described most system architectures as homogeneous entities. However, in general we can consider them each as levels in a hierarchy. For example, it may make sense to use FSMD (see FSMD subsection of Section 5.2.2) or vector coprocessor (see Vector coprocessors subsection of Section 5.2.4) nodes to implement the dataflow operators in a streaming dataflow system architecture (Section 5.2.1). Further, to model and coordinate changes in the composition of the dataflow network over time, it may make sense to model each of the dataflow configurations as a state in a very coarse-grained FSM (see Phased reconfiguration manager subsection of Section 5.2.2). With a variety of system architectures, rich implementation options within each, and their hierarchical compositions, we have a broad and powerful set of techniques to exploit the flexibility in reconfigurable computing platforms.

References

[1] M. Shaw, D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

[2] C. A. R. Hoare. *Communicating Sequential Processes*, International Series in Computer Science, Prentice-Hall, 1985.

[3] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 1967.

[4] E. A. Lee. The problem with threads. *IEEE Computer* 36(5), May 2006.

[5] S. Kleene. Recursive predicates and quantifiers. *Transactions of the American Mathematical Society* 53(1), 1943.

[6] A. M. Turing. On computable numbers, with an application to the entscheidungs problem. *Proceedings of the London Mathematical Society* 42(2), 1937.

[7] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics* 58, 1936.

[8] E. A. Lee, D. G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE* 75(9), September 1987.
[9] S. S. Bhattacharyya, P. K. Murthy, E. A. Lee. *Software Synthesis from Dataflow Graphs* (Synchronous Dataflow chapter), Kluwer Academic, 1996.
[10] T. M. Parks. *Bounded Scheduling of Process Networks*, UCB/ERL|95-105, University of California at Berkeley, 1995.
[11] Arvind, R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers* 39(3), March 1990.
[12] D. E. Culler, S. C. Goldstein, K. E. Schauser, T. von Eicken. TAM—a compiler-controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, June 1993.
[13] J. Hennessy, D. Patterson. *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, 2002.
[14] S. Devadas, Hi-K. T. Ma, R. Newton. On the verification of sequential machines at differing levels of abstraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 7(6), June 1988.
[15] J. Babb, M. Rinard, C. A. Moriz, W. Lee, M. Frank, R. Barua, S. Amarasinghe. Parallelizing applications into silicon. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999.
[16] E. Lee, A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 17(12), December 1998.
[17] E. Lee, S. Neuendorffer. Concurrent models of computation for embedded software. *IEEE Proceedings—Computers and Digital Techniques* 152(2), March 2005.
[18] D. Gajski, L. Ramachandran. Introduction to high-level synthesis. *IEEE Design and Test of Computers* 11(4), 1994.
[19] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers* 30(7), 1981.
[20] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*, MIT Press, 1986.
[21] R. Razdan, M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. *Proceedings of the 27th Annual International Symposium on Microarchitecture*, November 1994.
[22] S. Hauck, T. Fry, M. Hosler, J. Kao. The chimaera reconfigurable functional unit. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 1997.
[23] T. Callahan, J. Hauser, J. Wawrzynek. The Garp architecture and C compiler. *IEEE Computer* 33(4), April 2000.
[24] E. H. Baalbergen. Design and implementation of parallel make. *Computing Systems* 1(2), 1988.
[25] L. G. Valliant. A bridging model for parallel computation. *Communications of the ACM* 33(8), August 1990.
[26] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. Knight, Jr., A. DeHon. GraphStep: A system architecture for sparse-graph algorithms. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006.
[27] J. Wawrzynek, K. Asanovic, B. Kingsbury, J. Beck, D. Johnson, N. Morgan. Spert-II: A vector microprocessor system. *IEEE Computer*, March 1996.
[28] J. A. Jacob, P. Chow. Memory interfacing and instruction specification for reconfigurable processors. *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 1999.

[29] M. Gardner. The fantastic combinations of John Conway's new solitaire game "Life." *Scientific American* 223, October 1970.

[30] M. Wrighton, A. DeHon. Hardware-assisted simulated annealing with application for fast FPGA placement. *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 2003.

[31] N. Margolus. An FPGA architecture for DRAM-based systolic computations. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[32] T. Kobori, T. Maruyama, T. Hoshino. A cellular automata system with FPGA. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

CHAPTER 6

PROGRAMMING FPGA APPLICATIONS IN VHDL

Nachiket Kapre
Department of Computer Science
California Institute of Technology
André DeHon
Department of Electrical and Systems Engineering
University of Pennsylvania

Modern field-programmable gate arrays (FPGAs) contain hundreds of thousands of lookup tables (LUTs), hundreds of embedded memories, and hundreds of multipliers connected through a programmable interconnect fabric. Obviously it is intractable to program the FPGA at the granularity of these individual elements. However, with modern synthesis and layout tools, it is possible to describe a design simply by writing logical expressions, a level higher than gates, and letting the tools do the rest. Register transfer level (RTL) design is a popular discipline for describing these logical expressions. It allows the designer to express the design by describing the logic between each pair of register stages. This allows her to carefully control register-to-register logic depth while freeing her from selecting the actual gates and their mapping to the FPGA. Very High-Speed Integrated Circuit Hardware Description Language (VHDL) is one popular programming language that supports RTL hardware descriptions.

VHDL enjoys widespread popularity among designers in the industry, along with its close cousin, Verilog. Indeed, almost all modern CAD tools that perform simulation, synthesis, and layout support both. Verilog differs from VHDL primarily in the syntax it uses (VHDL is derived from Ada; Verilog, from C), but both languages are IEEE standards and are periodically reviewed to reflect changing industry realities and expectations.

VHDL is a strongly typed, Ada-based programming language that includes special constructs and semantics for describing concurrency at the hardware level. These concurrency constructs are new for most programmers and can be a source of confusion for beginners. In the following sections, we provide a tutorial overview of how to express and compose synchronous designs in VHDL. Through examples, we highlight the control one can exercise in VHDL to direct proper synthesis of hardware. We first look at how VHDL can be used to describe a design structurally as a composition of subcircuits. We then show how to express hardware in RTL form. Next we illustrate how hardware can be generated parametrically in a programmable

manner. Finally we outline the basic tool and workflow for developing VHDL designs.

This chapter is by no means a complete discussion of all VHDL language features. For a more comprehensive treatment of language syntax and coding style the reader is referred to the work of Ashenden [1, 2] and the appropriate vendor manuals (e.g., Xilinx, Inc. [3]).

6.1 VHDL PROGRAMMING

Programming in VHDL is quite different from programming in C because of its concurrent semantics. However, it does have several similarities with object-oriented languages like C++ and Java (e.g., encapsulation and interfaces). These common principles should help beginners understand the basic structure of the language and help them relate to hardware-specific VHDL constructs. In this section, we describe a few simple design elements in VHDL to outline key language features and illustrate important programming concepts.

We first show how to program a 2-input multiplexer using a structural abstraction. We then program a 4-input multiplexer using RTL semantics. Next we illustrate the use of parametric hardware generation by creating a 16-bit wide, 4-input multiplexer using a 1-bit, 4-input multiplexer from the previous example. Then we combine structural and RTL styles in a finite-state machine (FSM) datapath example to show how to use them in the same design. This final example introduces the programming of FSMs in VHDL.

6.1.1 Structural Description

To describe a multiplexer structurally, we first decompose it into primitive gates derived from its Boolean equations. Each gate is *instantiated* individually and then connected to others. We can think of a structural decomposition as a textual representation of a schematic or as subroutines in a conventional programming language such as C. As with schematic capture, a structural decomposition permits code for a recurring design element to be shared. This means that we can design an element once and instantiate it as many times as required. Unlike schematic capture, a textual structural description can be modified and updated easily with a text editor. Moreover, a hierarchical decomposition allows the designer to manage the complexity of a large hardware design by breaking it up into individual, manageable pieces. Listing 6.1 and Figure 6.1 illustrate the following important concepts.

Listing 6.1 ■ A structural 2-input multiplexer.

```
1 library ieee;
2 use ieee.std_logic_1164. all;
3
4 — this is the entity declaration for the 2-input mux
5 — it is a list of ports into the module.
```

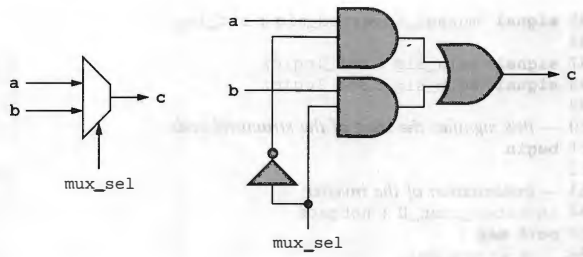


FIGURE 6.1 ■ A structural 2-input multiplexer.

```
 6 entity mux2 is
 7 port (
 8    a : in std_logic;
 9    b : in std_logic;
10    mux_sel : in std_logic;
11    c : out std_logic
12 );
13 end;
14
15 — this is where the structure of the multiplexer is defined
16 architecture struct of mux2 is
17
18 — all components that will be used in the structure
19 — need to be declared before use.
20 component notgate is
21 port (
22    a : in std_logic;
23    b : out std_logic
24 );
25 end component;
26
27 component andgate is
28 port (
29    a : in std_logic;
30    b : in std_logic;
31    c : out std_logic
32 );
33 end component;
34
35 component orgate is
36 port (
37    a : in std_logic;
38    b : in std_logic;
39    c : out std_logic
40 );
41 end compcnent;
42
43 — internal signals/wires used to connect the components
44 — also need to be declared here.
```

```
45 signal muxsel_inverted_sig : std_logic;
46
47 signal sela_sig : std_logic;
48 signal selb_sig : std_logic;
49
50 -- this signifies the start of the structural code
51 begin
52
53 -- instantiation of the inverter
54 inverter_inst_0 : notgate
55 port map (
56    a => mux_sel,
57    b => muxsel_inverted_sig
58 );
59
60 -- instantiation of the and gate
61 and_inst_a : andgate
62 port map (
63    a => a,
64    b => muxsel_inverted_sig,
65    c => sela_sig
66 );
67
68 -- another instantiation of the and gate
69 and_inst_b : andgate
70 port map (
71    a => b,
72    b => mux_sel,
73    c => selb_sig
74 );
75
76 or_inst : orgate
77 port map (
78    a => sela_sig,
79    b => selb_sig,
80    c => c
81 );
82
83 end;
```

1. VHDL files typically start by including the IEEE library and certain important packages like `std_logic_1164` (Listing 6.1, lines 1–2) that permit the use of type `std_logic` and Boolean operations on it. Additional packages such as `std_logic_arith` and `std_logic_unsigned` are often included for supporting arithmetic operations.

2. The VHDL description of a hardware module requires an `entity` declaration (Listing 6.1, lines 6–13) that specifies the interface of the module with the outside world. It is an enumeration of the interface ports. The declaration also provides additional information about the ports such as their direction (in/out), data type, bit width, and endianness. An entity declaration

in VHDL is analogous to an interface definition in Java or a function header declaration in C.

3. Almost all VHDL signals and ports use the data type `std_logic` and `std_logic_vector`. These data types define how VHDL models electrical behavior of signals, which we discuss in the Multivalued logic subsection of Section 6.1.5. The vector `std_logic_vector` allows declaration of buses that are bundled together. We will see its use in a subsequent example.

4. While an entity specifies the interface of a hardware module, its internal structure and function are enclosed within the `architecture` definition (Listing 6.1, lines 16–83).

5. In a structural description of a module, the constituent submodules are declared, instantiated, and connected to each other. Each submodule needs to be first declared in the `component` declaration (Listing 6.1, lines 20–25). This is merely a copy of the entity declaration where only the submodule's interface is specified. Once the components are declared, they can then be instantiated (Listing 6.1, lines 54–58). Each instance of the component is unique, and a component can have multiple instances (Listing 6.1, lines 61 and 69). The instantiated components are connected to each other via internal signals by a process called *port mapping* (Listing 6.1, lines 55–58). Port mapping is performed on a signal-by-signal basis using the => symbol. It is analogous to assembling a set of integrated circuits (ICs) on a breadboard and wiring up the connections between the IC pins using jumper wires. Observe the similarity between the schematic representation of the multiplexer and the structural VHDL in the example.

6. Notice in the example that the component for the AND gate is reused for each AND gate in the design (Listing 6.1, lines 61–66 and 69–74). This is one of the benefits of a structural representation—it permits reuse of existing code for recurring design elements and helps reduce total code size.

7. The submodules used in Listing 6.1 are primitives supported in the vendor library. In a larger design that is a collection of several multiplexers, the different multiplexers can be declared, instantiated, and connected to each other as required. A design can have several such levels of structural hierarchy. Hierarchy is a fairly common technique for design composition.

6.1.2 RTL Description

The multiplexer's RTL description can be specified much more succinctly than its corresponding structural representation. In RTL, logic is organized as transformations on data bits between register stages. By selecting the number of pipeline stages wisely, the designer can create a high-performance, high-speed hardware implementation, and by carefully deciding the degree of resource sharing, the size of the mapped design can be controlled as well. RTL provides the designer with sufficient low-level control to allow her to create an implementation that meets her specifications.

For the VHDL description, we still need the logical equations that define the multiplexer, but these can now be represented directly as equations, from

which a synthesis tool *infers* the actual gates. The tool tries to choose the gates on the basis of user-specified design criteria such as high speed or small area.

Listing 6.2 shows how to write a 4-input multiplexer with registered outputs (Listing 6.1 simply showed a 2-input multiplexer without a register).

1. As before, we start with the package and entity declarations (Listing 6.2, lines 6–18).

2. The RTL description of the VHDL entity is enclosed in the **architecture** block (Listing 6.2, lines 20–52). The logic equations and registers that are part of the RTL description are written here. Earlier, we used the **architecture** block to write the structural port-mapping statements.

Listing 6.2 ■ RTL for a 4-input multiplexer.

```
1 — library and package includes
2 library ieee;
3 use ieee.std_logic_1164.all;
4
5 — entity declaration for the 4-input multiplexer
6 entity mux4 is
7 port (
8    clk : in std_logic;
9    reset : in std_logic;
10   a : in std_logic;
11   b : in std_logic;
12   c : in std_logic;
13   d : in std_logic;
14   — notice the use of the type vector.
15   mux_sel : in std_logic_vector (1 downto 0);
16   e : out std_logic
17 );
18 end;
19
20 — RTL description of the multiplexer is defined here
21 architecture rtl of mux4 is
22
23 — internal signals used in the multiplexer are
24 — declared here before use
25 signal e_c : std_logic;
26
27 — indicates start of the actual RTL code
28 begin
29
30 — concurrent signal assignment
31 — the multiplexer functionality is described
32 — at a level above gates
33 e_c <= a when mux_sel="00" else
34    b when mux_sel="01" else
35    c when mux_sel="10" else
36    d;
37
38 — sequential signal assignment
```

```
39 process (clk, reset)
40 begin
41
42 — action under reset
43 if (reset = '1') then
44    e <= '0';
45 — action under rising clock edge
46 elsif (clk' EVENT and clk='1') then
47    e <= e_c;
48 end if;
49
50 end process;
51
52 end;
```

3. In the structural example, we saw how signals were used as wires for connecting component ports. In VHDL, signals are also used for representing logic. A signal can be defined as a function of one or more signals. The assignment operation is represented by the symbol <=, which is analogous to the = operation in C; however, the manner in which signals are assigned values is quite different from C.

4. As before, a signal needs to be declared before the **begin** statement (Listing 6.2, line 25). Each signal is defined using a signal assignment statement that describes the logic that drives it. A signal assignment statement can be either concurrent or sequential.

5. A concurrent signal assignment is used to describe the logic equation for the multiplexer (Listing 6.2, lines 33–36). Concurrent statements are written inside the **begin-end** statements of the **architecture** block but outside any **process** blocks (Listing 6.2, lines 39–50). For simulation purposes, a concurrent statement can be thought of as being evaluated in parallel with other concurrent statements.

6. In the listing, a sequential assignment describes a register (Listing 6.2, lines 39–50). The behavior of the register under reset and a rising edge of the clock is defined between the **begin-end** statements of the **process** block, which is itself enclosed within the **begin-end** statements of the **architecture** block (Listing 6.2, lines 21–52). A **process** block is executed only when any signal on its *sensitivity list* (e.g., `clk` and `reset` signals in Listing 6.2, line 39) changes value.

As their name suggests, sequential assignment statements enclosed within a **process** block are executed sequentially. A process is suspended when it finishes evaluating all of the statements it can inside the block, and signals are assigned values only at that time. Additionally, during evaluation of a **process** block, a signal retains the same logical value it had when the process began execution. This can be a potential source of confusion for new programmers. In Listing 6.6, we show how to write combinational logic using sequential statements.

7. Notice the compactness with which the multiplexer was described in Listing 6.2 (52 lines of RTL code versus 83 for structural). This is one of the key benefits of RTL over purely structural descriptions.

6.1.3 Parametric Hardware Generation

VHDL allows the designer to generate hardware as a function of some changeable parameter. This is a useful technique for code reuse when we need several variants of an element in the same design (e.g., an 8-bit and 16-bit adder in the same design). Certain design parameters are often not known until late in the design cycle, and some can change as the design specification evolves to meet customer requirements. It might also be necessary to perform a parametric design space exploration based on certain variables before deciding on the final architecture. These issues can be resolved with VHDL **generic**s.

The generics are specified at the start of the entity declaration. In the simplest form, VHDL allows the designer to write signals as vectors of parametric width. More advanced uses of parametric hardware generation employ **generate** statements, and **generate** loops can be used to create multiple copies of a repeating logic block.

In Listing 6.3 and Figure 6.2, we illustrate the use of parametric hardware generation using a multibit 4-input multiplexer. The width of the multiplexer is defined by a generic `DATA_WIDTH` (Listing 6.3, lines 8–13), which sets the range of the vectors in the interface and is later used as the termination value in the **generate** loop (Listing 6.3, lines 47–61). `DATA_WIDTH` copies of the 4-input multiplexer described in Listing 6.2 are instantiated and connected to the interface ports appropriately (Listing 6.3, lines 54–59).

Listing 6.3 ■ Parametric generation of a multibit 4-input multiplexer.

```
1  — library and package includes
2  library ieee;
3  use ieee.std_logic_1164.all;
4
5  — entity declaration of the multiplexer array
6  entity mux4_array is
7  — definition of the generic for this entity
8  generic (
9     — here 16 is the default value
10    — it can be redefined during
11    — instantiation, or during synthesis
12    DATA_WIDTH : integer := 16
13 );
14 port (
15    clk : in std_logic;
16    reset : in std_logic;
17    — notice the use of generic for constraining the vector length
18    a : in std_logic_vector(DATA_WIDTH-1 downto 0);
19    b : in std_logic_vector(DATA_WIDTH-1 downto 0);
20    c : in std_logic_vector(DATA_WIDTH-1 downto 0);
21    d : in std_logic_vector(DATA_WIDTH-1 downto 0);
```

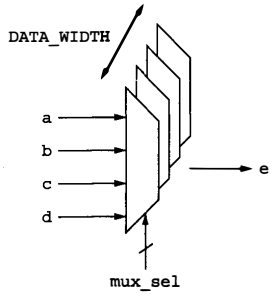


FIGURE 6.2 ■ Parametric generation of a multibit 4-input multiplexer.

```
22    mux_sel : in std_logic_vector(1 downto 0);
23    e : out std_logic_vector(DATA_WIDTH-1 downto 0)
24 );
25 end;
26
27 — the parametric code is enclosed within the architecture block
28 architecture parametric of mux4_array is
29
30 — like structural VHDL, the component being used needs to be declared here
31 component mux4 is
32 port (
33    clk : in std_logic;
34    reset : in std_logic;
35    a : in std_logic;
36    b : in std_logic;
37    c : in std_logic;
38    d : in std_logic;
39    mux_sel : in std_logic_vector(1 downto 0);
40    e : out std_logic
41 );
42 end component;
43
44 begin
45
46 — loop for generating a programmable number of mux4 instances
47 bitslices_gen : for i in 0 to DATA_WIDTH-1 generate
48    inst_mux : mux4
49    port map (
50       clk => clk,
51       reset => reset,
52       — notice the use of loop variable i for indexing
53       — into the array
54       a => a(i),
55       b => b(i),
56       c => c(i),
57       d => d(i),
58       mux_sel => mux_sel,
```

```
59        e => e(i)
60    );
61 end generate bitslices_gen;
62
63 end;
```

6.1.4 Finite-state Machine Datapath Example

In Listing 6.4, we design a time-shared datapath that computes Ax^2+Bx+C using only one multiplier and one adder. The design is naturally separated into state machine controller and datapath components. The controller and the datapath are designed using RTL and composed together structurally. The multiplier, the adder, and the associated multiplexers and registers are part of the datapath, whereas the control signals for the datapath multiplexers (Listing 6.4, lines 80–82) and registers (Listing 6.4, lines 84–86) are generated by the controller. Figure 6.3 shows the structural decomposition and the associated VHDL code. We can see that the control signals are connected from the controller to the datapath in the structural VHDL representation.

Listing 6.4 ■ A structural representation of the FSM datapath design.

```
1  — library and package includes
2  library ieee;
3  use ieee.std_logic_1164.all;
4  use ieee.std_logic_unsigned.all;
5
6  — entity declaration of the state-machine controller
7  entity fsm_datapath is
8  port (
9     — system signals
10    clk : in std_logic;
11    reset : in std_logic;
12
13    — input interface
14    start : in std_logic;
15    A : in std_logic_vector(3 downto 0);
16    B : in std_logic_vector(3 downto 0);
17    C : in std_logic_vector(3 downto 0);
18    x : in std_logic_vector(3 downto 0);
19
20    — output interface
21    output_valid : out std_logic;
22    result : out std_logic_vector(12 downto 0)
23 );
24 end;
25
26 architecture struct of fsm_datapath is
27
28 component fsm is
29 port (
30    — system signals
```

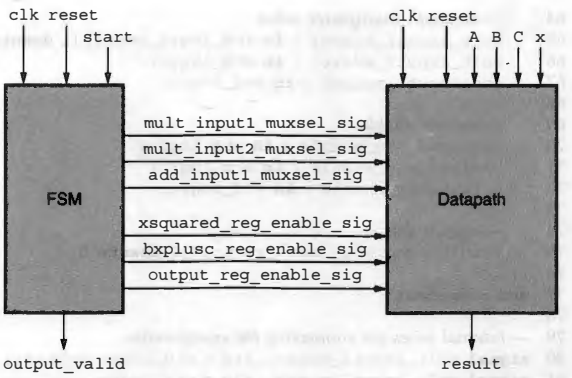


FIGURE 6.3 ■ A structural representation of the FSM datapath design.

```
31    clk : in std_logic;
32    reset : in std_logic;
33
34    -- start the computation
35    start : in std_logic;
36
37    -- datapath multiplexer select
38    mult_input1_muxsel : out std_logic_vector(1 downto 0);
39    mult_input2_muxsel : out std_logic;
40    add_input1_muxsel : out std_logic;
41
42    -- register enables
43    xsquared_reg_enable : out std_logic;
44    bxplusc_reg_enable : out std_logic;
45    output_reg_enable : out std_logic;
46
47    -- indicate output is valid
48    output_valid : out std_logic
49 );
50 end component;
51
52 component datapath is
53 port (
54    -- system signals
55    clk : in std_logic;
56    reset : in std_logic;
57
58    -- input operands
59    A : in std_logic_vector(3 downto 0);
60    B : in std_logic_vector(3 downto 0);
61    C : in std_logic_vector(3 downto 0);
62    x : in std_logic_vector(3 downto 0);
63
```

```
64     — datapath multiplexer select
65     mult_input1_muxsel : in std_logic_vector(1 downto 0);
66     mult_input2_muxsel : in std_logic;
67     add_input1_muxsel : in std_logic;
68
69     — register enables
70     xsquared_reg_enable : in std_logic;
71     bxplusc_reg_enable : in std_logic;
72     output_reg_enable : in std_logic;
73
74     — output data
75     result : out std_logic_vector(12 downto 0)
76  );
77  end component;
78
79  — internal wires for connecting the components
80  signal mult_input1_muxsel_sig : std_logic_vector(1 downto 0);
81  signal mult_input2_muxsel_sig : std_logic;
82  signal add_input1_muxsel_sig : std_logic;
83
84  signal xsquared_reg_enable_sig : std_logic;
85  signal bxplusc_reg_enable_sig : std_logic;
86  signal output_reg_enable_sig : std_logic;
87
88  — start component instantion and wiring
89  begin
90
91  datapath_inst : datapath
92  port map (
93
94     — system signals
95     clk => clk,
96     reset => reset,
97
98     — input operands
99     A => A,
100    B => B,
101    C => C,
102    x => x,
103
104    — datapath multiplexer select
105    mult_input1_muxsel => mult_input1_muxsel_sig,
106    mult_input2_muxsel => mult_input2_muxsel_sig,
107    add_input1_muxsel => add_input1_muxsel_sig,
108
109    — register enables
110    xsquared_reg_enable => xsquared_reg_enable_sig,
111    bxplusc_reg_enable => bxplusc_reg_enable_sig,
112    output_reg_enable => output_reg_enable_sig,
113
114    — output data
115    result => result
116 );
117
```

```
118 fsm_inst : fsm
119 port map (
120    — system signals
121    clk => clk,
122    reset => reset,
123
124    — start the computation
125    start => start,
126
127    — datapath multiplexer select
128    mult_input1_muxsel => mult_input1_muxsel_sig,
129    mult_input2_muxsel => mult_input2_muxsel_sig,
130    add_input1_muxsel => add_input1_muxsel_sig,
131
132    — register enables
133    xsquared_reg_enable => xsquared_reg_enable_sig,
134    bxplusc_reg_enable => bxplusc_reg_enable_sig,
135    output_reg_enable => output_reg_enable_sig,
136
137    — indicate output is valid
138    output_valid => output_valid
139 );
140
141 end;
```

We use the RTL form to describe the datapath, and we use a combination of concurrent and sequential statements for this purpose. The structure of the datapath is shown in Listing 6.5 and Figure 6.4.

Listing 6.5 ■ A time-shared datapath for computing $Ax^2 + Bx + C$.

```
1  — include the unsigned package to support arithmetic operations.
2  library ieee;
3  use ieee.std_logic_1164.all;
4  use ieee.std_logic_unsigned.all;
5
6  — describes the interface to the datapath,
7  — with its operands and control signals listed.
8  entity datapath is
9  port (
10    — system signals
11    clk : in std_logic;
12    reset : in std_logic;
13
14    — input operands
15    A : in std_logic_vector(3 downto 0);
16    B : in std_logic_vector(3 downto 0);
17    C : in std_logic_vector(3 downto 0);
18    x : in std_logic_vector(3 downto 0);
19
```

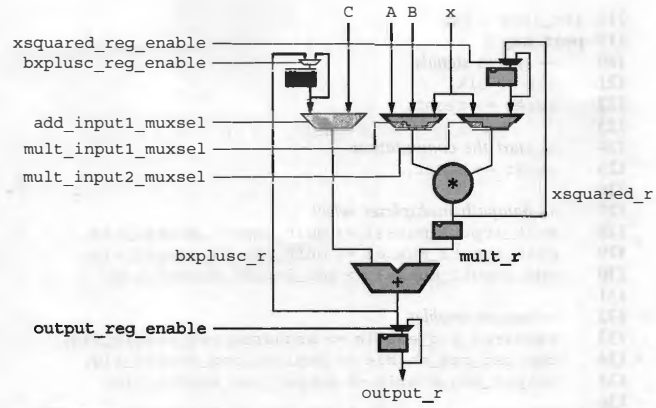


FIGURE 6.4 ■ A time-shared datapath for computing Ax^2+Bx+C.

```
20      — datapath multiplexer select
21      mult_input1_muxsel : in std_logic_vector(1 downto 0);
22      mult_input2_muxsel : in std_logic;
23      add_input1_muxsel : in std_logic;
24
25      — register enables
26      xsquared_reg_enable : in std_logic;
27      bxplusc_reg_enable : in std_logic;
28      output_reg_enable : in std_logic;
29
30      — output data
31      result : out std_logic_vector(12 downto 0)
32  );
33  end;
34
35  architecture rtl of datapath is
36
37  — notice the different bitwidths on each signal
38  — these precisions have been carefully selected
39  — based on the multiply/add operations and input
40  — bitwidths
41  signal mux_0_c : std_logic_vector(3 downto 0);
42  signal mux_1_c : std_logic_vector(7 downto 0);
43  — mux_1_c needs 8 bits of precision due to
44  — x-squared at the input
45  signal mux_2_c : std_logic_vector(8 downto 0);
46  — mux_2_c needs 9 bits of precision due to
47  — precision of Bx+C
48
49  signal mult_c : std_logic_vector(11 downto 0);
50  — product of 8-bit and 4-bit inputs is 12-bit
51
```

```
52  signal add_c : std_logic_vector(12 downto 0);
53  -- sum of 12-bit and 9-bit inputs is 13-bits with overflow
54
55  signal mult_r : std_logic_vector(11 downto 0);
56  signal output_r : std_logic_vector(12 downto 0);
57  signal bxplusc_r : std_logic_vector(8 downto 0);
58  signal xsquared_r : std_logic_vector(7 downto 0);
59
60
61  begin
62
63  -- concurrent statements to describe the multiplexers
64  mux_0_c <= A when mult_input1_muxsel = "00" else
65             B when mult_input1_muxsel = "01" else
66             x;
67
68  mux_1_c <= "0000"&x when mult_input2_muxsel = '0' else
69             xsquared_r;
70
71  mux_2_c <= "00000"&C when add_input1_muxsel = '0' else
72             bxplusc_r;
73
74  -- multiplier
75  mult_c <= mux_0_c * mux_1_c;
76
77  -- adder
78  -- the extra 0s at the MSB of the inputs are
79  -- to capture overflow bit in the result
80  add_c <= ("0000"&mux_2_c) + ('0'&mult_r);
81
82  -- define all registers
83  all_registers : process(clk, reset)
84  begin
85
86  if (reset= '1') then
87
88     mult_r <= (others=>'0');
89     xsquared_r <= (others=>'0');
90     bxplusc_r <= (others=>'0');
91     output_r <= (others=>'0');
92
93  elsif (clk' EVENT and clk='1') then
94
95    -- infer simple register
96    mult_r <= mult_c;
97
98    -- notice that we are not specifying
99    -- the else condition. the synthesis tool will
100   -- infer a latch for this case. if enable is
101   -- low, previous value will be retained.
102   if (xsquared_reg_enable='1') then
103      xsquared_r <= mult_c(7 downto 0);
104   end if;
105
```

```
106    if (bxplusc_reg_enable='1') then
107       bxplusc_r <= add_c(8 downto 0);
108    end if;
109
110    if (output_reg_enable='1') then
111       output_r <= add_c;
112    end if;
113
114 end if;
115 end process;
116
117 -- drive the output with a simple wire from the register
118 result <= output_r;
119
120 end;
```

Included in this datapath design is the special package `std_logic_unsigned` (Listing 6.5, line 4), which allows us to express arithmetic operations using high-level symbols (+ and *) on signals of type `std_logic_vector`. These functions are defined in the package. The package also helps us infer the right kind of arithmetic units (e.g., signed or unsigned). VHDL supports the signed data type for arithmetic operations.

Notice that we must carefully specify the precision required for all internal signals (Listing 6.5, lines 37–58). We must also pad extra 0s when the input signal precision is smaller than that of the operator (Listing 6.5, lines 68–69). The concatenation operator & in VHDL further allows us to combine the right mix of signals to enter the datapath as required by the design. This low-level control makes VHDL suitable for designers seeking to customize their designs to the problem.

We represent the multiplexers, multipliers, and adders using concurrent statements (Listing 6.5, lines 63–80), which are evaluated in parallel and inferred as combinational logic blocks. Note that all three multiplexers evaluate their inputs simultaneously. Concurrent statements allow the designer to capture this hardware-level concurrency in VHDL. Also note, however, that there is a dataflow dependency between the multiplexers and the multiplier (as well as the multiplexer and the adder). These dependencies are converted into wires that connect the appropriate logic blocks together, but each logic block continues to evaluate its inputs in parallel. The dataflow dependency only means that signal changes are propagated to the downstream multiplier input after a suitable delay for the multiplexer evaluation (see Delta delay subsection of Section 6.1.5 for more information on this delay).

We express the registers in the design using sequential statements inside the **process** block (Listing 6.5, lines 83–115). Most registers have a conditional signal assignment (Listing 6.5, lines 102–104). Notice the absence of an **else** statement or a default value on the rising clock edge. This implies that the signal retains its previous value if the condition for assignment is not

satisfied. VHDL automatically infers feedback from the output to the multiplexer at the register input. If the **else** is present or if a default value is specified, no feedback will be inferred. This can be seen in Listing 6.6 (signals in the next-state decoder process have default values, avoiding inference of feedback paths).

To design the state machine controller, we first create a time sequence of operations that must be performed to obtain the final result. This gives us a cycle-by-cycle schedule for how the datapath elements are shared between the different operations. Each of these cycles is represented by a state, which is then decoded into multiplexer select and register enable signals for the datapath. The VHDL for this state machine is written in an RTL form specialized for state machines. It is shown in Listing 6.6 and illustrated in Figure 6.5.

Listing 6.6 ■ A state machine for generating control signals for the time-shared datapath.

```
1  — library and package includes
2  library ieee;
3  use ieee.std_logic_1164.all;
4
5  — entity declaration of the state-machine controller
6  entity fsm is
7  port (
8     — system signals
9     clk : in std_logic;
10    reset : in std_logic;
11
12    — start the computation
13    start : in std_logic;
14
15    — datapath multiplexer select
16    mult_input1_muxsel : out std_logic_vector(1 downto 0);
17    mult_input2_muxsel : out std_logic;
18    add_input1_muxsel : out std_logic;
19
20    — register enables
21    xsquared_reg_enable : out std_logic;
22    bxplusc_reg_enable : out std_logic;
23    output_reg_enable : out std_logic;
24
25    — indicate output is valid
26    output_valid : out std_logic
27 );
28 end;
29
30 — state-machine code is enclosed is defined inside this architecture block
31 architecture behav of fsm is
32
33 — define an enumerated type for state
34 type state_type is (IDLE, COMPUTE_BX, COMPUTE_BXPLUSC_AND_XSQR,
35                     COMPUTE_AXSQR, COMPUTE_ASQRPLUSBXPLUSC, ASSERT_OUTPUT);
```

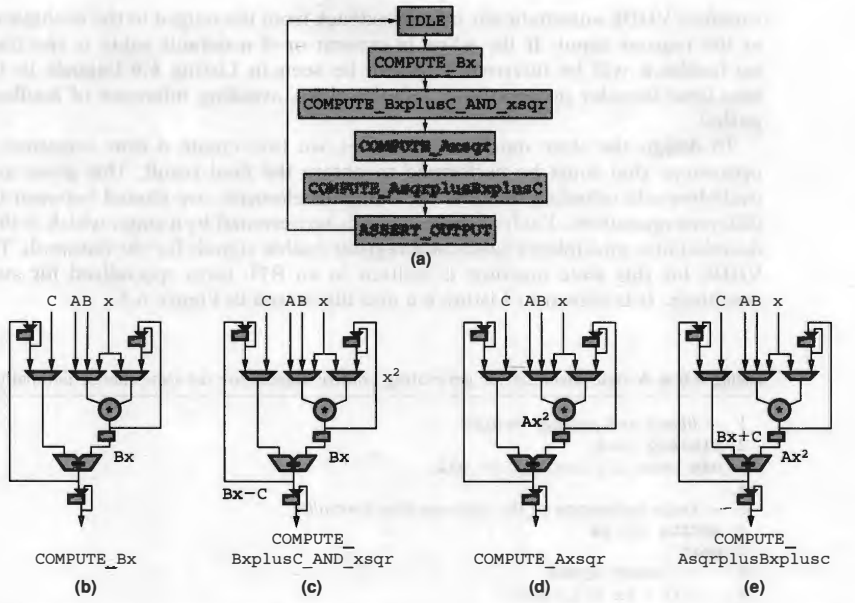


FIGURE 6.5 ■ A state machine for generating control signals for the time-shared datapath. Labels on wires show dataflow steps in calculation.

```
36  signal state_c : state_type;
37  signal state_r : state_type;
38
39  — internal signals
40  signal mult_input1_muxsel_c : std_logic_vector(1 downto 0);
41  signal mult_input2_muxsel_c : std_logic;
42  signal add_input1_muxsel_c : std_logic;
43  signal xsquared_reg_enable_c : std_logic;
44  signal bxplusc_reg_enable_c : std_logic;
45  signal output_reg_enable_c : std_logic;
46  signal output_valid_c : std_logic;
47
48  — start the signal assignments
49  begin
50
51  — logic to compute the next state of the state machine
52  — also generate the control signals [only combinational, right now]
53  next_state_decoder : process(state_r, start)
54  begin
55
56  — given initial values for all signals
57  mult_input1_muxsel_c <= "00";
58  mult_input2_muxsel_c <= '0';
```

```
59  add_input1_muxsel_c <= '0';
60  xsquared_reg_enable_c <= '0';
61  bxplusc_reg_enable_c <= '0';
62  output_reg_enable_c <= '0';
63  output_valid_c <= '0';
64  state_c <= IDLE;
65
66  — specify state transistions
67  — update state variable
68  — update the control signals
69  case state_r is
70    when IDLE =>
71
72       — conditional state transition
73       if (start='1') then
74
75          state_c <= COMPUTE_BX;
76
77          mult_input1_muxsel_c <= "01"; — select B
78          mult_input2_muxsel_c <= '0'; — select x
79
80    end if;
81
82    when COMPUTE_BX =>
83
84    — unconditional state transition
85    state_c <= COMPUTE_BXPLUSC_AND_XSQR;
86
87    mult_input1_muxsel_c <= "10"; — select x
88    mult_input2_muxsel_c <= '0'; — select x
89    xsquared_reg_enable_c <= '1'; — save x*x
90    bxplusc_reg_enable_c <= '1'; — save Bx+C
91    add_input1_muxsel_c <= '1'; — select C
92
93  when COMPUTE_BXPLUSC_AND_XSQR =>
94
95    state_c <= COMPUTE_AXSQR;
96
97    mult_input1_muxsel_c <= "00"; — select A
98    mult_input2_muxsel_c <= '1'; — select xsqr
99
100 when COMPUTE_AXSQR =>
101
102   state_c <= COMPUTE_ASQRPLUSBXPLUSC;
103
104   add_input1_muxsel_c <= '1'; — select Bx+C
105   output_reg_enable_c <= '1';
106
107 when COMPUTE_ASQRPLUSBXPLUSC =>
108
109   state_c  <= ASSERT_OUTPUT;
110   output_valid_c <= '1';
111
112 when ASSERT_OUTPUT =>
113
```

```
114    state_c <= IDLE;
115
116  end case;
117
118  end process;
119
120  -- describe the registers that hold the state bits
121  -- the actual bits will be inferred by the
122  -- synthesis tool from the symbolic states
123  state_register : process(clk, reset)
124  begin
125
126  if (reset = '1') then
127    state_r <= IDLE;
128  elsif (clk' EVENT and clk='1') then
129    state_r <= state_c;
130  end if;
131
132  end process;
133
134  -- register the control signals generated during state transitions
135  output_logic : process(clk, reset)
136  begin
137
138  if (reset = '1') then
139
140     mult_input1_muxsel <= "00";
141     mult_input2_muxsel <= '0';
142     add_input1_muxsel <= '0';
143     xsquared_reg_enable <= '0';
144     bxplusc_reg_enable <= '0';
145     output_reg_enable <= '0';
146     output_valid <= '0';
147
148  elsif (clk EVENT and clk='1') then
149
150    mult_input1_muxsel <= mult_input1_muxsel_c;
151    mult_input2_muxsel <= mult_input2_muxsel_c;
152    add_input1_muxsel <= add_input1_muxsel_c;
153    xsquared_reg_enable <= xsquared_reg_enable_c;
154    bxplusc_reg_enable <= bxplusc_reg_enable_c;
155    output_reg_enable <= output_reg_enable_c;
156    output_valid <= output_valid_c;
157
158  end if;
159
160  end process;
161
162  end;
```

By encoding the state of the controller with an enumerated data type (Listing 6.6, lines 34–36), we can defer the actual encoding of the state bits until the synthesis stage. The synthesis tool then assigns a bit encoding to optimize logic. It is easier to verify the operation of the state machine using

symbolic states. It is also easier to update and modify symbolic state machine code.

In the next-state decoder (Listing 6.6, lines 53–118), we enumerate all possible states of the state machine and define state transitions from each of them. These transitions are expressed as conditions under which the state changes.

We use the **process** block for describing purely combinational logic in the next-state decoder of the state machine (Listing 6.6, lines 53–118). Previously, we used **process** for describing only registers (Listing 6.2, lines 39–50). This shows how we can write combinational logic here as well. In this listing, notice that the same signal is assigned values multiple times in the **process** block (signal `mult_input1_muxsel_c` in Listing 6.6, lines 57, 77, 87, and 97). As the statements are evaluated sequentially, the last signal assignment statement to be evaluated is considered valid, superseding all previous assignments. During execution of sequential statements in a process, for purposes of determining new signal values all signals are considered to have the same value they had at the start of the process. Signals that are assigned values inside the process will acquire those values only when process execution is complete—that is, **process** suspends. It is in this aspect that the VHDL sequential semantics are different from those of a conventional programming language (e.g., C). Figure 6.6 shows similar code written in C and VHDL to illustrate how the different execution semantics lead to different answers.

In Listing 6.6, all signals are assigned a value at the beginning of the process. By design, only one **when** subblock of the **case** statement will be evaluated, which means that only those signals that have assignments inside the valid **when** subblock will get new values (Listing 6.6, line 77, 87, or 97 will execute; line 57 will execute in all cases). According to the VHDL sequential signal assignment rule, these new assignments will hold when the process suspends. Other signals will simply carry the default values they were assigned at the start. This avoids the inference of feedback that we saw earlier (refer to Listing 6.2).

```
1 process(clk)
2 begin
3
4       if(clk 'EVENT'and clk='1') then
5               counter <= counter + 1;
6               if(counter=10)
7                       counter <= 0;
8               end if;
9       end if;
10
11 end process;
12
13 — if counter=9 at start of process,
14 — when process suspends, counter=10.
```

(a)

```
1 int updatecounter(int counter){
2       counter++;
3
4       if(counter==10)
5          counter = 0;
6
7       return counter;
8 }
9
10 // updatecounter(9) returns 0
```

(b)

FIGURE 6.6 ■ Comparison of sequential VHDL (a) and C (b) assignment semantics.

6.1.5 Advanced Topics

Delta delay

VHDL uses an event-driven simulation model. A signal is evaluated only when an event—that is, a signal transition associated with the input signals—has occurred. Once a statement is evaluated, its associated signal needs to be assigned the newly generated value. However, this is not done right away so as to keep the evaluations of other statements from using this new value immediately, potentially leading to inconsistent results.

Remember that in VHDL all concurrent statements are evaluated in parallel. Hence, to keep the simulation consistent VHDL uses delta delay, in which the newly generated value is scheduled as an event at the following delta. (A delta is simply a logical delay used in the simulator and not a physical delay of the circuit.) The simulator will generate as many deltas as required depending on the logical depth of the circuit and its input transitions. Once all events for a given delta are exhausted, the simulator proceeds to the earliest delta at which the next event exists. Physical time in the simulator is advanced only when no more events are left to be processed at the last delta at the current physical time. Sometimes the simulator is unable to advance its physical time because of asynchronous, combinational feedback loops that continue generating new events at incremental deltas. Such loops should be avoided when programming VHDL, and modern synchronous simulation and synthesis tools usually warn the designer if such a loop is detected.

Multivalued logic

Another electrical behavior is modeled in VHDL using the multivalued logic type `std_logic`. It allows a signal to have different kinds of electrical states, apart from a Boolean 0 or 1, which are required for modeling tristate drivers, multiple simultaneous drivers (usually a design error), uninitialized signals, and weak drivers.

6.2 HARDWARE COMPILATION FLOW

To fully understand how VHDL fits into the design process, we expand the FPGA compilation process shown in Figure I.2. Our flow is shown in Figure 6.7.

1. The hardware designer begins the design-engineering process with a problem specification—that is, a functional description of the problem along with additional performance and area constraints that the implementation must meet.

2. Based on this specification and the inherent problem structure, the designer identifies an appropriate system architecture to use for the implementation. We saw different kinds of system architectures in Chapter 5.

3. The designer writes VHDL code to describe this design using structural and RTL styles that we saw earlier in this chapter.

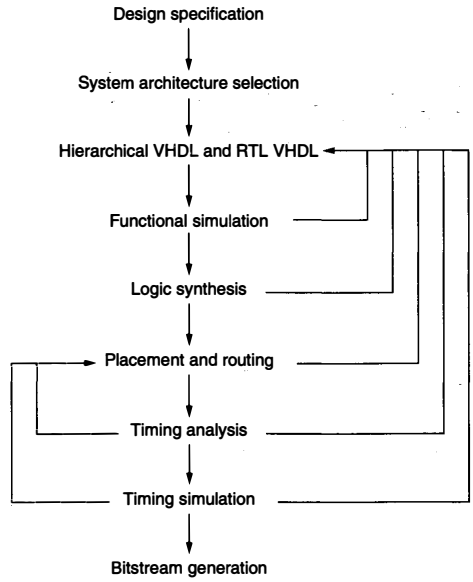


FIGURE 6.7 ■ FPGA compilation flow.

4. Once the VHDL is written, the designer needs to first check if her VHDL meets functional specifications, using a suitable testbench that can be written in VHDL itself. The testbench and the design are run in a logic-level simulator. The testbench generates appropriate test vectors for the design and verifies the result. This is typically an iterative process, and the designer continues to refine the VHDL design until the functional specification is met.

5. After verifying correctness, the designer then proceeds to the FPGA back-end phase, a multistage (and iterative) process. It starts with synthesis, where the synthesis tool converts the VHDL description of the design (excluding the testbench) into a logic-level FPGA netlist. This netlist is generated by first inferring hardware from VHDL code and then optimizing it through several state-of-the-art algorithms—for example, logic minimization, retiming (Chapter 18), covering (Chapter 13), and sharing to meet timing and area constraints. Constraints can be specified as a separate input to the tool by the user.

6. The designer uses backend tools to perform placement (Chapter 14) and routing (Chapter 17) on the synthesized logic elements to map them to an actual physical device (logic elements are assigned physical LUTs while the

wires between them are mapped to the interconnect fabric). This is typically the most time-consuming step of the backend process. The designer can help direct these tools using additional constraints (see Section 6.2.1) either to improve the quality of the final mapped design or to reduce the compilation time needed.

7. Once the design is placed and routed, the designer can perform static timing analysis to ensure that the timing constraints are met. FPGA tools can also write out a post place and route timing annotated VHDL netlist for a timing simulation that models logic and interconnect delays accurately. Specific timing requirements not covered in the simple static timing analysis can then be simulated and checked.

8. If the designer is satisfied with the performance of her implemented hardware, the tools generate a programming file for the FPGA device (Chapter 19).

6.2.1 Constraints

Constraints are an indispensable tool directive that a designer can use to help her designs meet required specifications. They can be used to direct the synthesis tools in optimizing the design for either high-speed operation or low-area implementation (these are usually conflicting goals). For example, the designer can specify a frequency target that Synplify Pro (a synthesis tool) must meet using the following timing constraint.

```
set option -frequency 300.000
```

This sets the target frequency for the compilation to be 300 MHz. Similarly, designers can provide timing constraints for the placement and routing phases as well.

```
TIMESPEC "clock signal name"=3.3ns;
```

More important, a designer can give physical floorplanning constraints to direct the placement and routing algorithms to use a specified region on the chip.

```
INST "*" AREA GROUP = "dummy name";
AREA GROUP "dummy_name" RANGE = SLICE X0Y0:SLICE X100Y100;
```

Here we create a group `dummy_name` containing all hardware elements in the design using wildcards (`*`). Then we specify a rectangular box from 0,0 to 100,100 on the FPGA. The units are measured in `SLICE`s; a `SLICE` is a cluster of a few, usually four, Xilinx FPGA LUTs. The proper selection of these constraint values is typically based on intuition and can be refined with designer experience. Placement constraints, such as the one in the previous code snippet, are vendor and device specific, but each vendor typically has analogous constraints for each device.

6.3 LIMITATIONS OF VHDL

Although VHDL currently enjoys a healthy market share, there are several limitations and drawbacks in the language:

- VHDL syntax is verbose, extremely cumbersome, and requires several lines of code to describe even simple logic elements (e.g., a register typically requires four to ten lines of code).
- Hardware needs to be described at a very low level of abstraction (i.e., RTL). The programmer is responsible for specifying the logic that goes between each register stage, which can become a significant programming challenge for large irregular designs with thousands of registers and unique logic between register stages.
- As technology and FPGA architectures evolve, the optimal amount of pipelining required to meet the desired cycle time changes. Because RTL is written for a specific number of registers in the logic path, it needs to be rewritten when the number of register stages changes. In other words, the amount of logic between register stages must be modified accordingly.
- Low-level descriptions also make it hard for synthesis tools to optimize and schedule logic. Programmer bias disallows optimizations that might have otherwise been possible in a more flexible description.
- Hardware described in VHDL suffers from the additional drawback of significantly long verification times. It is known that equivalent simulation-specific, cycle-accurate models written in C, C++, Java, or other higher-level language can be simulated 10 to 100 times faster than in VHDL. Verification is a significant portion of the design cycle, and there is demand to contain the time spent on it.

In subsequent chapters, we will see other high-level languages that address many of these limitations (e.g., Chapters 7, 9, and 10). In many cases, however, these languages use VHDL as an intermediate target in their mapping flow.

References

[1] P. Ashenden. *The Designer's Guide to VHDL*, 2nd ed., Morgan Kaufmann, 2002.
[2] P. Ashenden. *The Student's Guide to VHDL*, Morgan Kaufmann, 1998.
[3] *Development System Reference Guide*, Xilinx, Inc.

CHAPTER 7

COMPILING C FOR SPATIAL COMPUTING

Timothy J. Callahan
School of Computer Science
Carnegie Mellon University
André DeHon
Department of Electrical and Systems Engineering
University of Pennsylvania

This chapter describes techniques for compiling from C or similar languages to reconfigurable architectures. We will first briefly describe the benefits of this approach and the contexts where it is most useful. Then we will describe in detail the algorithms and their technical limitations and challenges.

For the discussion in this chapter, we assume the presence of a microprocessor coupled with the reconfigurable fabric (RF). This eases adaptation in several ways and is particularly useful when supporting a mix of irregular control tasks (best suited to the microprocessor) and compute-intensive, high-throughput tasks (best suited to the RF), as described in the Processor subsection of Section 5.2.2.

The original C code can be partitioned between the central processing unit (CPU) and the RF at several granularities, including procedures, compound loops, inner loops, and blocks. The algorithms described in this chapter apply to any of these cases. The appropriate granularity for a particular system will depend on the hardware available and the particular costs involved in communication between the CPU and the RF and will not be treated in this chapter.

For most of this section we will assume that the source code, both before and after the designer's target-specific efforts to improve performance via hints in comments or pragmas, will be legal C code as defined by the ISO standard [9]. However, at the end we will overview some methods for integrating blocks designed via HDL or schematic capture into a C program.

The benefits to having a full, pushbutton path that starts from C and that can put at least some of the application on the reconfigurable hardware follow.

- There are many more C programmers than hardware designers, and writing an algorithm in C is typically faster than in an HDL.
- There is a large existing code base even for embedded applications, with at least the reference version written in C.
- Working with a single description of the entire program makes it easy for the designer or compiler to quickly explore the tradeoffs of different hardware/software partitionings. Also, it allows both hardware (HW) and

software (SW) versions to be created so that the operating system can choose at runtime which is better (see Chapter 11).

- Designers can start with automatic compilation, and then focus their efforts on improving a few loops while benefiting from the compiler's speedup on the remainder. Furthermore, with the compiler's support the designer's required effort is reduced in many cases to simply restructuring the code or embedding simple compiler directives in the form of comments or `#pragma` syntax.
- The code can be easily tested on a conventional microprocessor for correctness.

This chapter will be of direct value to those interested in compilation for spatial computing from a sequential language. More generally, it will give an application writer an understanding of the power and limitations of the state of the art of such compilers—and thereby how to write high-performance code quickly.

7.1 OVERVIEW OF HOW C CODE RUNS ON SPATIAL HARDWARE

This section provides a quick overview of how C code can be implemented on a reconfigurable fabric. It assumes basic familiarity with C. The approaches used are simple and far from optimal, but easy to understand. The detailed algorithms of how a compiler does this construction will follow.

In the figures that follow (e.g., Figure 7.1), the gray rectangles represent registers. For simplicity, the global clock is not shown. An arrow from the side toward the register indicates a load enable signal. The hardware appears at the operator level, not at the gate/CLB level.

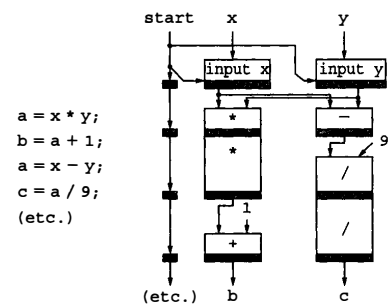


FIGURE 7.1 ■ Straight-line code.

7.1.1 Data Connections between Operations

The simplest components of C code to start with are sequences of straight-line arithmetic and logical statements. A sequence effectively tells us the set of primitive operations that make up the computation and how those operations are linked together—that is, they tell us how the outputs of one operation become inputs to other operations.

In a C program, the statements execute in order. A statement can define a variable, and subsequent statements using that variable get its last defined value. This is how value definitions are connected to their use(s)—the most recent assignment to a variable is the one that is used by a subsequent statement.

With spatial computation, each operation is implemented as a function unit (or *module*) and a producer is connected to its consumer(s) by a direct physical connection. Even if two different C statements assign to the same program variable, they are treated as different variables internally. In the example in Figure 7.1, the two definitions of variable *a*, while sequential in the C program, are actually independent and can be performed in parallel spatially. This is one step in the direction of exploiting the unlimited parallelism of spatial hardware, where we wish to reduce unnecessary ordering of operations as much as possible and keep only the necessary ordering.

Because we are implementing the computation spatially and in parallel, the actual compute datapaths are always instantiated, ready to perform their operations. It is sometimes necessary to inform the modules when their inputs are available and when they should actually perform their actions. The chain of registers on the left of Figure 7.1 acts as a very simple sequencer. In this particular example, the registers simply count off how many cycles are required to compute all of the results. A '1' bit is fed from the `start` signal, kicking off the sequencer and latching values into the input modules. The input modules hold the input values constant during execution of this unit of computation. When a 1 bit appears at `finish`, the final values are ready to pass on.

Mixed operations of different complexity (e.g., adders and multipliers) may take different amounts of time to complete. For efficient operation, rather than slowing all operators down to the latency of the slowest one, it is often worthwhile to decompose slower operators into multiple cycles, potentially pipelining them internally. In this example, multiply and divide are split into two stages requiring two cycles, while add and subtract require just one cycle each.

Throughout this section, we employ a timing discipline where values are held constant until the end of their block schedule. If a module's output register is shown at level P in the schedule, and the overall schedule length is SL, then the output of that module is guaranteed to be correct and stable from cycles P through SL of that specific block execution (where cycle 0 is when the `start` signal is raised).

7.1.2 Memory

Memory loads and stores pose additional complications beyond simple arithmetic and logical operations, in that their effects are not just local. In particular,

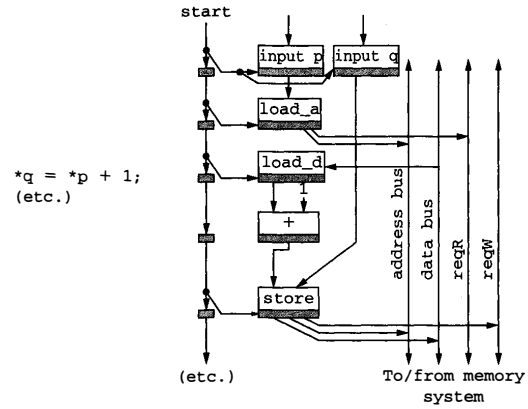


FIGURE 7.2 ■ Implementation of memory accesses.

memory can be used to perform dynamic interconnect between operations, and we must be careful to preserve the original communication semantics of the C program. A "memory" function unit has local input and output connections to other function units as normal, but also has connections to global shared address, data, and control buses. These connect each memory node to the same shared memory system.

Memory access operations must be scheduled on a particular cycle both to allow sharing among memory operations and to preserve sequential C semantics. Without scheduled coordination, two modules can attempt to drive the address or data bus simultaneously. The simple controller triggers each memory access at the correct time so that no clashes arise on either the address or the data buses. Memory access must be scheduled after its input values are ready. The compiler is also responsible for scheduling memory accesses in a way that ensures that each pair that might access the same memory location is performed in the correct relative program order.

The example in Figure 7.2 shows how a load node is split into a `load_a`, which sends the address and load request, and a `load_d` (or *load continuation*), which grabs the data when it comes back. The example assumes a load latency of just one cycle. If the memory system takes extra time to return the load data, as in the case of a cache miss, there must also be a `stall` signal factored into the sequencer to freeze execution of the subcircuit; this is *not* shown in the figure.

7.1.3 If-then-else Using Multiplexers

Simple if-then-else statements can be merged into a single subcircuit by performing the operations along both branches and then using multiplexers to

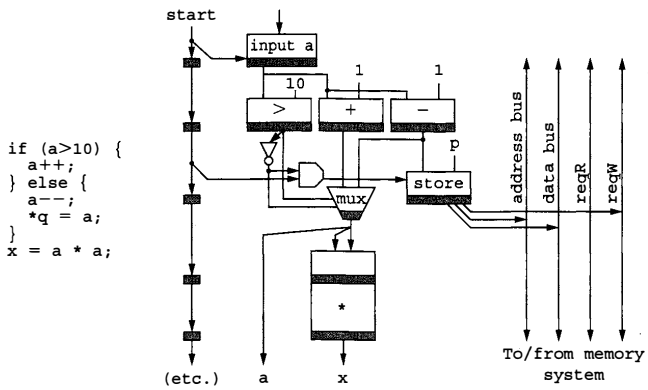


FIGURE 7.3 ■ If-conversion: Combining if-then-else using predicates and multiplexers.

select the correct version of each variable for use in subsequent computation. This removes the branch; instead, the comparison result is used as a *predicate* to choose the correct variable for later use, as with variable `a` in the example in Figure 7.3. In the figure the predicates are the result of the comparison `a > 10` and its inverse, which say whether the `then` or the `else` branch is taken. In general, a predicate is always a Boolean value—the result of a comparison, or a Boolean function of multiple comparisons, as occurs when nested if-then-else statements are reduced. `switch` statements and even forward `goto` statements can be implemented using similar techniques.

If the `then` or `else` contains a side-effect-causing operation, such as the `store` in Figure 7.3, that operation's cycle trigger must be ANDed with the predicate under which it should execute.

7.1.4 Actual Control Flow

To map C code containing more than just simple if-then-else control flow to the reconfigurable fabric, some real control flow is needed. Control flow means that there may be multiple subcircuits on the RF; only one is active at a time; and the transition from one to another subcircuit is guided by the values that are computed by the ongoing computation. This is spatial computation's implementation of a conditional branch.

The control flow is implemented with the control bit: When it reaches the end of a subcircuit, it is directed to the start of the next subcircuit to execute. When a subcircuit has multiple successors, a predicate controls which one receives the control bit. In Figure 7.4, we see the explicit branch either to a subcircuit performing the `then` computation or to the one performing the `else` computation. Subcircuit SC1 computes the condition `a > 10`, and the result determines

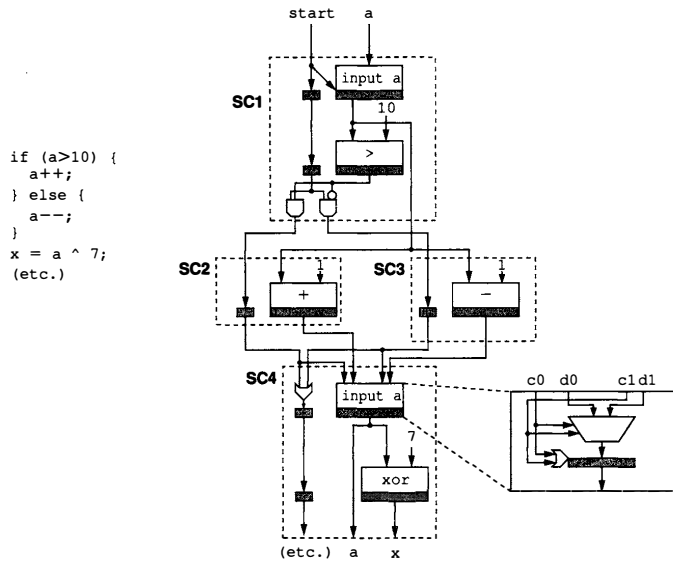


FIGURE 7.4 ■ Actual control flow.

whether the control bit goes to SC2 or SC3; then one or the other gets the control bit and executes. Control flow paths then merge at SC4, where a control bit from either SC2 or SC3 starts SC4's execution. Note that the source of the control bit entering SC4 also controls whether SC2's or SC3's final version of `a` is latched at the start of SC4 (note in Figure 7.4 the expansion of `input a`).

Subcircuits as small as those shown in Figure 7.4 would not typically be created by the compiler; instead, they would likely be merged as shown earlier. However, if SC2 and SC3 had very different execution lengths, it would be worthwhile to keep them separate like this. If, for example, one had 1-cycle latency and the other 13-cycle, we would only experience the 13-cycle latency when that path was taken. In contrast, when uneven paths are combined into one subcircuit, we pay the worst-case latency every execution.

A subcircuit that has a single predecessor actually does not require input modules, assuming in our implementation that the predecessor subcircuit holds its outputs constant until it is activated again. This simplification is shown in SC2 and SC3 of Figure 7.4.

A loop is implemented simply by control branching back to the top of itself or to some other, earlier subcircuit.

7.1.5 Optimizing the Common Path

We have seen two extremes: (1) combining all the computation in an if-then-else nest and (2) doing no combining and keeping all branches. But the key to getting the best performance from limited spatial hardware is *selectively* merging the computation on the common path(s) (to remove the subcircuit-to-subcircuit latency and to expose operation parallelism) while excluding computation on the rarely taken paths (so that it doesn't get in the way of the common case).

In Figure 7.5 we see the same code as in Figure 7.4, but we have merged the computation along the path with the increment. However, we have excluded the path with the decrement. The compiler chose to merge the computation along the path with the increment (SC1 → SC2 → SC4 from Figure 7.4) into one subcircuit because a test run (or the programmer) told it that that path was more commonly executed. Because reentering the merged increment path is not allowed, we needed to copy the XOR computation for the decrement path.

Merging the common path allowed the compiler to schedule the comparison and the addition in parallel, reducing computation time to three cycles. The schedule for the common case is also better than that for the case where all blocks were merged, as in that case we needed a multiplexer to merge the results from the decrement path, and that would add an extra step between the addition and the XOR. In the general case, the benefit of excluding a rare path could be even greater: Consider if the decrement were instead a multiplication, or even a

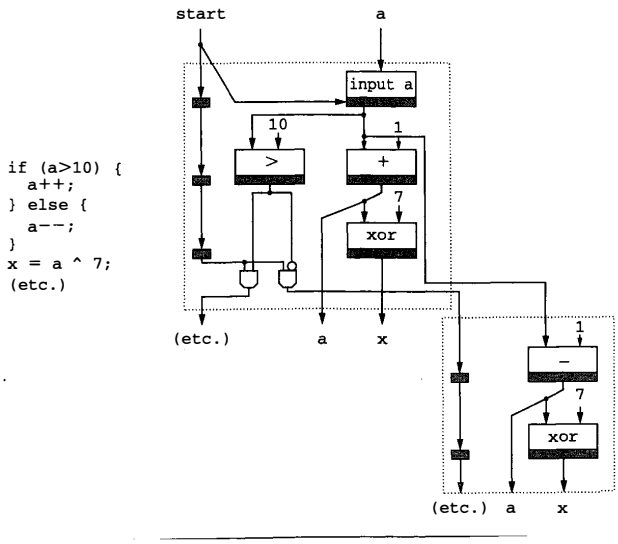


FIGURE 7.5 ■ Optimizing the common path.

long chain of operations. In that case, if that rare path were included, it would force a much longer schedule.

In this case, when the execution flow exits the common path and continues to the excluded path, the total time will be five cycles, longer than the four cycles that would have resulted if decrement had been included. Many 3-cycle executions with a few 5-cycle executions are better than all 4-cycle executions—again, optimizing the common path.

A system might also choose to implement rarely taken paths as normal software on the CPU. This would ease the demand for resources on the reconfigurable fabric and allow implementation of a loop or procedure that otherwise would not fit. This approach is also beneficial when the excluded path includes an operation, such as a library call, that cannot be implemented directly on the RF. However, the cost of transferring control to the CPU for a rare path, when it does happen, must be considered.

7.1.6 Summary and Challenges

In this section we sketched how C can be implemented spatially and began to illustrate optimizations for parallelism that are the key to extracting high performance from spatial hardware, even when the spatial hardware runs at a slower clock rate than the CPU. We also illustrated context-specific optimization, which allows us to highly specialize the computation to the common case execution of the application, further increasing parallelism and reducing the computation required. Nonetheless, these simple techniques leave us with spatial designs that can be inefficient and that underutilize our reconfigurable fabric. These inefficiencies include:

- *Not pipelining:* Sequential paths prevent us from reusing our spatial hardware at its full capacity; spatial operators sit idle for most of the cycles in a block. To fully use the capabilities of the reconfigurable hardware, datapaths should be pipelined for rapid reuse.
- *Memory:* Sequential dependencies among memory access operations limit available parallelism.
- *Operator size and specialization:* The reconfigurable fabric can provide hardware tailored to the compute needs (e.g., just the right datapath width, specialized around compile time constants), but specific information about operator size is often not immediately apparent in the original C program.

The following sections show how we can address many of the simple translation scheme's limitations.

7.2 AUTOMATIC COMPILATION

A particular compiler flow is largely determined by the system architecture. Here we will assume that fairly large pieces of code will be migrated

to the reconfigurable fabric—a loop or perhaps even a complete procedure. There is little difference in the algorithms between granularities at this level.

We assume a standard C compiler frontend that parses the source files (see Figure 7.6(a)) and performs further processing until the intermediate representation consists of a *control flow graph* (CFG) for each procedure. A CFG consists of *basic blocks*, each containing an ordered list of simple instructions and connected by control edges indicating a possible branch from the end of one basic block to the start of another, as shown in Figure 7.6(b). By definition, entry to a basic block occurs only at the beginning, exits occur only at the end, and all instructions inside the basic block execute once the block is entered.

Within each basic block, complex expressions are broken up by introducing compiler temporary variables so that each simple instruction contains just one operation. This list of simple instructions in each basic block resembles assembly code to some degree, but is of a higher level: variables (including compiler temporaries) are used instead of explicit registers, and all type information is still available. Many optimizations are performed on this representation to reduce the number of instructions by, for example, constant propagation, constant folding, and common subexpression elimination. (See Aho et al. [1] or Muchnick [13] for related background.)

The frontend also provides some standard analyses. Of particular interest here is live variable analysis, which indicates whether or not the current contents of a variable need to be preserved for a possible future use.

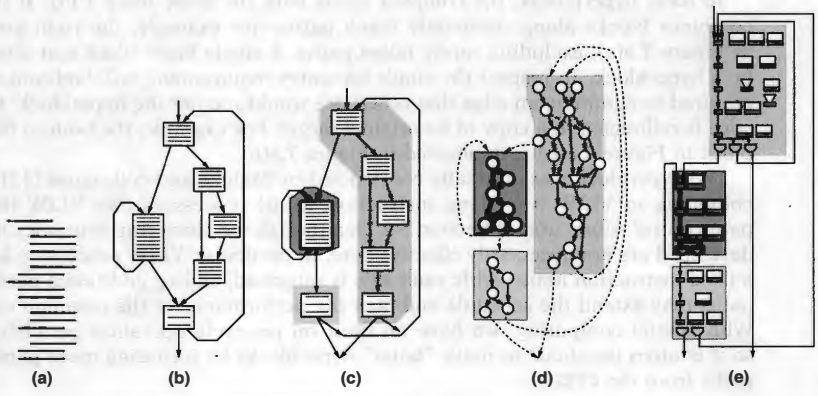


FIGURE 7.6 ■ Overall compiler flow: (a) original C source code, (b) CFG basic blocks, (c) clustering of basic blocks into hyperblocks, (d) construction of the DFG, and (e) circuit generation from the DFG.

After frontend processing produces an optimized CFG for each procedure, we start compilation steps specific to reconfigurable computing:

- *HW/SW partitioning:* This is very system dependent, and its discussion is deferred until section 7.3.1.
- *HW/HW clustering of the CFG basic blocks into hyperblocks:* illustrated in Figure 7.6(c) and discussed in section 7.2.1.
- *Building the dataflow graph (DFG) for each hyperblock:* illustrated in Figure 7.6(d) and discussed in section 7.2.2.
- *DFG optimization:* discussed in section 7.2.3.
- *Generating the circuit from the DFGs:* This involves module mapping (packing one or more DFG nodes into a single-cycle macro function unit), scheduling, connecting hyperblock subcircuits, and other related tasks; illustrated in Figure 7.6(e), which leaves out data connections, and discussed in section 7.2.4.

After we go over these steps, we will describe some uses and variations in Section 7.3.

7.2.1 Hyperblocks

Because basic blocks are limited to straight-line control flow between branches, they are often quite small and limit our opportunities for parallelism. As we saw in the previous section, we can often convert if-then-else constructs into dataflow using multiplexers. These composite blocks, or *hyperblocks*, have a single entry point at the top and one or more exits. All branches within the hyperblock are eliminated by using predicates and multiplexers. Each hyperblock becomes a subcircuit, as shown earlier.

To form hyperblocks, the compiler starts with the basic block CFG. It then combines blocks along commonly taken paths—for example, the right group in Figure 7.6(c), excluding rarely taken paths. A single basic block can always be a hyperblock. To respect the single top-entry requirement, *tail duplication* is required to eliminate an edge that otherwise would reenter the hyperblock; that edge is redirected to a copy of its original target. For example, the bottom basic block in Figure 7.6(b) is duplicated in Figure 7.6(c).

The hyperblock was originally constructed by Mahlke and colleagues [12] for compiling to VLIW (very long instruction word) processors (see VLIW datapath control subsection of Section 5.2.2), although the clustering heuristics they developed are not necessarily effective here. In particular, VLIW processors have a fixed instruction issue width; once this is saturated, adding additional parallel paths may extend the schedule and hurt the performance of the common case. With spatial computing, we have no limit on per-cycle operation parallelism, so it is often beneficial to make "fatter" hyperblocks by including more parallel paths from the CFG.

7.2.2 Building a Dataflow Graph for a Hyperblock

Here we focus on constructing a DFG (dataflow graph) from the set of basic blocks in a hyperblock. The DFG is a "stepping stone" between the original

software specification and the final spatial hardware implementation. The compiler performs many important tasks in building it:

- Control dependence within the hyperblock is converted to data dependence: Internal conditional branches are eliminated through the introduction of predicates (Boolean values indicating the "taken" path through the computation). The only remaining conditional branches are exits out of the hyperblock.
- Data producer–consumer relationships are made explicit via data edges in the graph; also, because a new DFG node is created for each definition, variable renaming is effectively performed, which eliminates false dependencies.
- Any remaining ordering constraints between individual operations, particularly memory operations, are also made explicit through ordering edges.

These actions convert the sequential ordering of instructions to a partial order of DFG nodes, exposing parallelism. In addition, maximal control speculation is employed so that all safe operations execute every iteration, removing dependencies between predicate calculations and those operations, breaking critical paths, and further increasing operation parallelism. Finally, the DFG is an ideal representation with which to perform many additional optimizations, described next.

The DFG is composed of nodes and edges:

- *Nodes:* These include constants, inputs to the hyperblock, simple computational operations having no side effects (such as addition), memory accesses, and exit nodes. Exit nodes are associated with an outgoing control edge from one hyperblock to another; when an exit node's predicate input is true, it causes a control transfer to the target hyperblock recorded on the node. The exit node also defines which live data values should be transferred to the successor hyperblock, as indicated by liveness edges.
- *Edges:* These are directed edges between the nodes and are of three types: data edges, indicating producer–consumer relationships; ordering edges, indicating an ordering constraint between two nodes such as memory operations; and liveness edges. Liveness edges go only to exit nodes. They indicate the set of values that are live-out at that hyperblock exit and thus must be copied out—that is, transferred to the successor hyperblock or back to the CPU. Each liveness edge is annotated with the name of the variable because, in general, the variable cannot be deduced from the source DFG node (a single node may be the source for different variables at different exits). These edges are necessary because the set of live variables to be transferred typically differs at each exit. Also, the source DFG node for a given variable can be different at different exits.

Top-level build algorithms

We build the DFG from the basic block CFG for each hyperblock. The algorithm for building the DFG performs a single forward pass, visiting each basic

block in the hyperblock in an order such that each basic block is visited only after all its predecessors have been visited. Then, when visiting each basic block, the simple instructions are visited in sequence. This forward pass builds all of the DFG nodes, including nodes directly translated from instructions as well as predicate calculation nodes and mux (multiplexer) nodes inserted to implement predicated execution. The forward pass also builds all data and ordering edges.

Building data edges

When a node is constructed, the compiler creates data edges to its inputs using the `lastDefs` data structure. Throughout the forward pass, this table is kept up to date regarding which node produced the last definition of each variable; there is at most one such definition at any point. We show an example in Figure 7.7.

At the start of processing a hyperblock's entry basic block, the `lastDefs` list is initialized with an input node associated with each live variable, as with `y:n1` in the example.

Whenever an instruction assigns to a variable, `lastDefs` is updated. In our example, `y++` in BB1 uses the current value of `y`, `n1` as the source for the incoming edge to the new add node, `n4`; then the `lastDefs` list is updated so that the new value of `y` is available from `n4`.

A copy—an assignment from one variable to another—requires no action other than updating the `lastDefs` list (see for example `x=y` in BB1 in Figure 7.7). A new entry for `x` is made in the `lastDefs` list, `x:n1`, just using the current entry for `y`. Similar for `z=y`, although at that point the entry for `y` is different so a different source node is given to `z`. This has the effect of performing

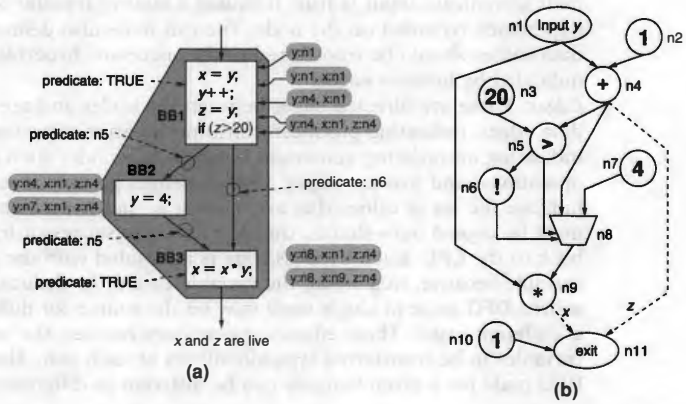


FIGURE 7.7 ■ Basic blocks selection for the hyperblock: (a) the state of the `lastDefs` list at various points in the process; (b) the resulting DFG.

copy propagation and constant propagation for free while building the DFG. At the end of processing each basic block, the final `lastDefs` list is recorded.

For a nonentry basic block B with a single predecessor in the hyperblock, the predecessor's final `lastDefs` list is used as the starting `lastDefs` list for processing B. This occurs from the end of BB1 to the start of BB2.

Building muxes

At a basic block with $N > 1$ incoming CFG edges, a given variable may have differing definitions arriving via the edges as indicated by the predecessors' respective final `lastDefs` lists. In such cases, an unencoded mux is constructed in the DFG to route the appropriate definition to subsequent consumers. An unencoded mux has N data inputs and N Boolean select inputs—only one of the select inputs can be true—and the corresponding data input is routed to the output. The N data inputs to the mux are from the data source nodes from the arriving `lastDefs` lists; the select input corresponding to each of the N data inputs is the predicate for that arriving edge. The data output of the mux structure becomes the definition of the variable entered in the `lastDefs` list for the start of processing that basic block. This occurs for `y` entering BB3, where the compiler inserts mux `n8` to select between sources `n4` and `n7`, and then makes `n8` the new entry for `y`. Because the entries for `x` and `z` are the same, however, no mux is built for either of them.

Predicates

At the beginning of processing each basic block, a node calculating that block's predicate is built if necessary and the predicate source is recorded to be used as input for nodes that cannot be executed speculatively (e.g., stores). The predicate for the hyperblock entry block is TRUE. For each other basic block, the predicate is built as the OR of the predicate sources of all incoming edges. When there is just one incoming edge, the calculation degenerates to just using that edge's predicate.

At the end of processing a basic block, a predicate is built if necessary and recorded for each outgoing edge. For a basic block ending in a conditional branch, an edge's predicate is built as its source block's predicate, ANDed with the branch condition under which that edge is taken. For a basic block ending in an unconditional branch, the edge predicate on the single outgoing edge is just the same as the block's predicate. After forming predicates for a nested if-then-else, it may be possible to simplify them; for example, a block may be (p1 AND p2) OR (p1 AND not p2), which can be reduced to just (p1) by rules of Boolean logic.

Ordering edges

To help build ordering edges, the compiler maintains lists of all loads and stores seen along any path from the entry of the hyperblock to the current point. At the start of processing the hyperblock, the lists are initialized as empty. At the end of processing each basic block, the state of the lists at that point is recorded. At the start of any nonentry basic block, the starting lists are

simply calculated: For a basic block with a single predecessor, the predecessor's lists are copied; when there are multiple predecessors, the respective lists are unioned.

When building a new load, construct an ordering edge from each upstream store to the new load, and then the load is added to the `seen_loads` list. When a new store is built, an ordering edge is constructed from each node on both the `seen_loads` and `seen_stores` lists to the new store and the store is added to the `seen_stores` list. This step is very conservative; for example, it adds an ordering edge from a store to each subsequent load even if the load is from a different array. Later phases use dependency information to remove ordering edges that are not necessary—that is, when it is guaranteed that the two accesses cannot refer to the same memory location.

Live variables at exits

This phase determines, for each exit, which values must be copied out to the next hyperblock or CPU when that exit is taken. For each such variable, a liveness edge is constructed from the node responsible for the last definition, as found in the `lastDefs` list, to the DFG exit node.

If the variable is live at that exit, there will be an entry for it in `lastDefs` at the point of exit. The indicated DFG node is the one providing the value for the variable, so the edge is constructed from that node to the exit DFG node.

Figure 7.8 shows an example of a swap. There are two exits from the first hyperblock, at one of which `a` and `b` are swapped—this results purely from

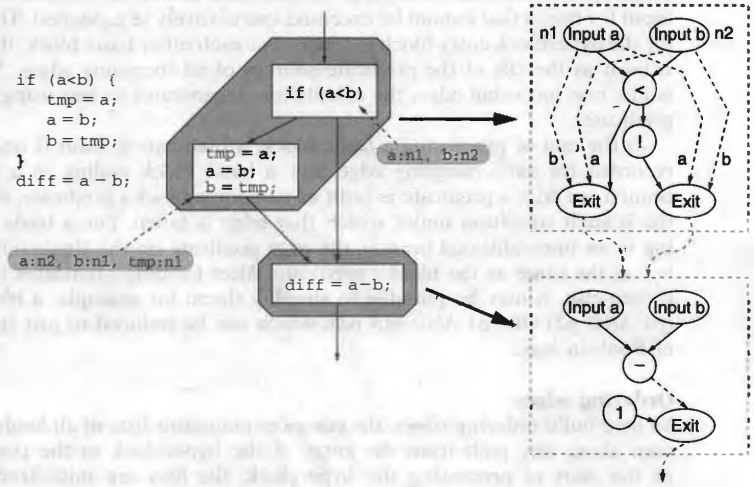


FIGURE 7.8 ■ Code, hyperblock formation, and resulting DFGs.

lastDefs list processing. The figure shows the differing contents of the lastDefs lists at the different exits. In one case, a's source is n1 (input a); in the other, its source is n2 (input b). Later, when the compiler translates the DFGs to subcircuit implementations, it will also form connections from the appropriate liveness edge sources in the first hyperbock to the input nodes in the second hyperblock.

Scalar variables in memory
If the address of a scalar variable is taken at some point by the C language & operator, it may be written or read through a pointer access. In this case, in general the variable must reside in memory. When direct accesses to the variable are interspersed with pointer accesses, we can't be sure when the pointer access might be accessing that variable without further analysis. Thus, we must keep the memory version of the variable up to date. When this situation occurs, each use of the variable requires an explicit load from memory, and each definition requires a store. Going to memory for each variable access is obviously detrimental to performance, especially on a reconfigurable fabric, so later optimizations attempt to eliminate or reduce the number of such accesses.

7.2.3 DFG Optimization

Optimizations have been performed by the compiler frontend before DFG construction even starts. More optimizations are performed during construction, some of them coming automatically in the construction process, such as constant and copy propagation. Finally, after the DFG is completed, the compiler performs many optimizations, often performing the same ones multiple times, and sometimes iterating a set of different optimizations until no further improvement occurs. We will review a few of these optimizations in the following subsections. (More detail can be found in other references; see the work of Budiu [4] and Callahan [5].) These optimizations consider the scope of the DFG (i.e., each hyperblock), which is larger than each basic block but smaller than the entire procedure.

Constant folding
Constant folding is simply the reduction of expressions of compile time constants to the equivalent constants. Its most obvious benefit is that it removes operations from the DFG and ultimately reduces area and latency in the subcircuit. A second benefit is that constant folding can enable operator specialization for other operations. (See Chapter 22.)

Figure 7.9 shows a simple example of constant folding. The important part of this example is observing how this opportunity for optimization occurs only after hyperblock formation, because the definition of x in B3 no longer interferes with constant propagation and constant folding in B1-B2-B4. This effect is not limited to constant folding, but has the potential to improve all optimizations described here.

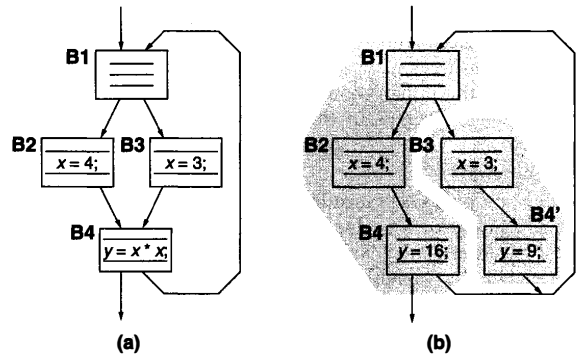


FIGURE 7.9 ■ The commonly taken path in the loop is B1-B2-B4 (a). Hyperblock formation (b)—tail duplication occurs with basic block B4. This enables constant propagation and then constant folding for the expressions `x*x`, although this is actually done after conversion to the DFG.

Identity simplification
This can be considered a special case of constant folding, that is, finding cases where the operator can be eliminated because one of the inputs is a *specific* constant. Integer operations that add or subtract zero, shift by zero, or multiply by one are eliminated. Similar optimizations exist for Boolean predicate operations: If either an OR or an AND has a constant input, it can be eliminated by replacing it either with a constant or with a pass-through from the other input.

Strength reduction
This replaces one operator with another operator (or operators) having less overall latency/area. For example, replace `x*2` with `x+x` or `x<<1`. Again, this is often based on having a specific constant input. Sometimes, equivalent implementations occur whether we do operator-level strength reduction or bit-level specialization, but it does not hurt to have multiple attacks. Multiplication by a constant is an important example because it occurs so often and because a general multiplication function unit can be expensive on a reconfigurable fabric! The expression `x*7` can be expressed as `(x<<2)+(x<<1)+x`, but even better as `(x<<3)-x`.

Dead node elimination
A cleanup pass eliminates nodes that are "dead"—that is, those that are not "live." A node is live when it is required for proper execution if (1) it has side effects (i.e., it is a store or an exit), or (2) its data output is used by another "live" node, including the case where the node supplies a live-out value to an exit node. The algorithm starts by marking as live all nodes with side effects: stores and exits. Then it marks as live any node whose data output is used by any other "live" node, and so on. Only data and liveness edges need to be traversed.

Once no more nodes can be marked as live, any remaining nodes not marked as such are known to be dead and can be safely removed.

Common subexpression elimination
Common subexpression elimination (CSE) is a well-known optimization for identifying and removing redundant computation—that is, the same operation is performed on the same operands. When a node has the same operands as another, it is immediately obvious from the structure of the graph. All simple operator nodes are subject to elimination, as are all nodes introduced to support predicated execution (Boolean calculations and muxes). Store and exit node types are not considered for elimination. Loads can be considered if additional analysis is done (see Memory access optimization subsection later).

Boolean value identification
The C language defines signed and unsigned integer data types of various sizes, but ISO C does not contain a Boolean data type [9]. Although the result of a comparison is defined to be either 0 or 1, the type of the result is a signed integer—typically 32 bits. However, no information is lost if only a single bit is used to carry the result. This can be exploited to advantage in hardware. Therefore, it is useful to identify as "Boolean" those operations guaranteed to produce only 0 or 1. When necessary for non-Boolean uses, Boolean values can be converted back to standard C type by zero-padding.

The algorithm identifies "base case" Boolean-producing nodes: comparisons, constant 0, and constant 1. Then it forward-propagates the Boolean property to nodes that have an opcode that preserves the Boolean property and that also have all inputs already flagged as Boolean. Opcodes that preserve the Boolean property include bitwise AND, OR, and XOR, as well as muxes. Opcodes that do not preserve the Boolean property include bitwise NOT and addition. However, all predicate calculations are marked as Boolean when they are constructed, including NOT operators.

For a compilation flow that eventually goes through commercial logic synthesis tools, many of the excess bits being trimmed would be trimmed eventually anyway. However, if the compiler needs to make decisions based on hardware area estimates—for example, for hardware/software partitioning—it is useful to have more accurate information about required bus and function unit width earlier in the compiler flow. This is also a motivation for the next two analyses.

Type-based operator size reduction
ISO C semantics [9] dictate that arithmetic and logical operations involving type `char` and/or `short` operands must be performed at the precision of type `int`. Figure 7.10 shows the implicit type conversions.

During initial DFG construction, all three casts are faithfully translated to DFG nodes. But since the destination's representation size of `short` (say 16 bits) is less than that of `int` (say 32 bits), the upper bits of the addition are discarded. Thus, a 16-bit adder will give the same result as a 32-bit adder in all cases, so in the intermediate representation we can signify that just a 16-bit adder

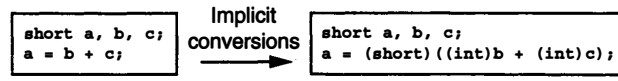


FIGURE 7.10 ■ Implicit type conversions.

is required. This in turn means that the addition uses just the lower 16 bits of each operand; thus, reducing the size of one operator may enable the size reduction of others. There also may be type conversions on the operands that can be eliminated, as shown in the figure. Besides the obvious savings in area and operator size for the addition, there are additional savings in this example: eliminating the two sign-extending type conversions on `b` and `c`.

Dataflow analysis-based operator size reduction
More detailed dataflow analyses can be performed to find the number of bits actually required by variables and operators. They may be based on range—for example, `i` within the loop `for (i = 0; i < 100; i++)`. They may also be bit level: propagating forward information about bits fixed at `0` or `1` and propagating backward information about bits not used (e.g., Budiu et al. [3]).

Memory access optimization
The handling of memory access ordering occurs in three phases:

1. The compiler conservatively adds ordering edges between pairs of memory accesses during DFG construction.
2. After DFG construction, the compiler tries to find and remove false ordering edges. Considering each pair of memory accesses connected by an ordering edge, it applies a series of tests. If any test can prove that the two operations can never access the same location during the same iteration, that ordering edge is removed. These various tests are based on array index analysis, pointer analysis, and simple testing of fixed locations (e.g., `&a` and `&b`).
3. Although removing false ordering edges is useful in itself because it exposes more parallelism and typically results in a shorter schedule, there are also many optimizations based on ordering edges that will see improved results.

Space does not allow the description of all memory optimizations that have been developed (see Callahan [5] or Budiu and Goldstein [2] for more examples), so just one will be presented here as an example.

Removing redundant loads
Consider this simple C code snippet:

```
a = *p;
*q = b;
c = *p;
```

Originally, there will be ordering edges from the first load to the store and from the store to the second load. But if subsequent pointer analysis can guarantee

that p and q can never point to the same location, those ordering edges will be removed.

The existence or absence of ordering edges is then used in the following optimization. Two loads can be reduced to one if (1) they definitely access the same location, and (2) there is no intervening store that might modify that location. Both of these requirements can be determined directly from the DFG.

To check (1), the compiler checks if the addresses of the two loads come from the same node (this assumes that common subexpression elimination has been run, which would ensure that equivalent addresses come from the same node). To check (2), we need to check for an intervening store. If there is a path from one of the loads to any store, and from that store to the other load, via ordering edges, then that store is intervening and represents a possible modification of that memory location. If both requirements hold—(1) same location and (2) no intervening store—then one of the loads can be eliminated, and its consumers can use the output of the other load. In this example, the store to *q was originally intervening, but is no longer after removal of the ordering edges.

7.2.4 From DFG to Reconfigurable Fabric

At this point we have an optimized DFG for each hyperblock. The final translation involves mapping DFG nodes to modules, scheduling each module to a specific timestep, and creating the simple sequencer, resulting in an actual subcircuit (RTL HDL description) for each hyperblock. Then, finally, connections are made among the sequencers and modules from different hyperblock subcircuits to complete the overall circuit.

Packing operations into clock cycles

A CPU cannot exploit the fact that a simple logical AND requires much less latency to complete than an integer addition; both take one cycle. But with spatial computing, we can pack multiple low-latency operations into a clock period (i.e., between registers) [6]. A typical example is predicate calculation, which consists of 1-bit Boolean calculations—a large subgraph of these can be performed in the time it takes to do one 32-bit addition. Another case is two successive ripple-carry adders because the latencies of their carry chains largely overlap. Additional opportunities arise from the context-specific optimization of each operation allowed by spatial computing (Chapter 22), which can greatly reduce the latency of a specific operation. On the other hand, long latency operations, such as multiplication, are typically split into stages across multiple cycles, and these stages are not considered for combining as noted before.

For simplicity it is useful to assume a target clock period from the start to get an even "packing," even if the reconfigurable platform supports a variable clock period. For systems with a fixed clock period, the upper bound is a hard limit. If the final circuit has a combinational path with latency exceeding the clock period, then some portion of the design flow must be rerun, either with more conservative decisions (for example, with operation packing) or with higher priority given to the failing paths. With a variable clock period, mistakes can be accommodated.

After this grouping, rather than a graph of operator nodes, we have a graph of modules, each of which implements one or more original DFG nodes (or a stage of a multi-cycle operation). Each module has a register at its output.

Scheduling

Scheduling a module-mapped DFG is straightforward using list scheduling. The output of list scheduling is, for each module m, an assigned slot $\sigma(m)$ when it *starts* computing. A module m's outputs are available to other modules starting at $\sigma(m)+\text{lat}(m)$, where $\text{lat}(m)$ is the latency (in clock cycles) of m (a multi-cycle operation is scheduled as a unit). In most cases this latency is one clock.

List scheduling maintains three lists of modules, and each module is a member of exactly one list. The three lists are:

- `scheduled`: modules that have already been assigned a slot. This is initialized to the input modules, all scheduled at slot 0.
- `ready`: modules whose sources have all been scheduled.
- `notready`: modules that have one or more sources not yet scheduled.

Then the list-scheduling algorithm iterates as follows until all modules have been scheduled:

1. Choose a module m from the `ready` list based on some priority heuristic.
2. Set S to the earliest cycle on which m can be scheduled, considering only when m's inputs are first all available.
3. If m has a resource conflict at slot S with any already scheduled module, increment S and go to step 3.
4. Schedule m in slot S and put it on `scheduled`.
5. Check m's successors and move them as appropriate from `notready` to `ready`.
6. If any nodes remain on `ready`, go to step 1.

Only memory operations can encounter a resource conflict in step 3, arising from the use of shared address and/or memory data buses. In contrast, any simple (nonmemory) module is scheduled as soon as all its inputs are available. Note that most such simple modules are not "actively" scheduled—they don't have an activation input from the sequencer. These passive modules simply compute a result each cycle whether or not their inputs are valid. After scheduling, the total schedule length is known, so the sequencer can be built to count off the cycles and trigger those modules that need it. The output of the final sequencer stage is ANDed with the predicate values for each exit node to create the appropriate outgoing control bit. Also, the source of each liveness edge to each exit node is translated to the appropriate connection to the input module in the destination subcircuit.

Pipelined scheduling

Here we will briefly give an idea of how pipelined scheduling works. Only hyperblocks branching to themselves to form a self-loop are considered. In the

final implementation, the key difference is that with pipelined scheduling the calculation of the control bit that is fed back to the top of the sequencer is produced not at the end of the schedule but somewhere in the middle. The result is that there are multiple '1' control bits shifting through the sequencer simultaneously, corresponding to the fact that multiple iterations of the loop are executing in an overlapped fashion. The compiler must now watch out for resource conflicts between successive iterations when scheduling the loop. The spacing between successive iterations is limited by either loop-carried data or memory dependencies, or by resource requirements. Further details are available in works by Callahan [5, 8].

Connecting memory nodes to the memory ports
Recall that each load node is split into a `load_a`, for sending the request and address, and a `load_d`, for receiving the data. Our circuit diagrams have implied that shared access to the memory port uses buses driven by tristate buffers, which some FPGAs have. But this approach could run out of tristate buffers or could restrict placement options. An alternative is to use an unencoded mux to drive each input to the shared port. For example, a mux might replace the address bus; when a memory module asserts a request to its control line of the mux, its address is routed to the mux output and to the memory port. The load data bus returning data from memory does not need any active routing; it is driven only by the memory port and fans out to all of the `load_d` modules, one of which will latch the result. However, additional buffering may be required to avoid timing problems when fanout is large.

What next?
Although we have shown the implementations as schematics, what we actually have at this point is a structural (RTL) description in an HDL such as Verilog or VHDL (Chapter 6). In a system with a commercial FPGA as its reconfigurable fabric, there is likely a fixed wrapper circuit that handles the details of connections between the compiler-generated circuit and the FPGA pins connected to the CPU and external memory. The wrapper and compiled circuit together are fed through commercial tools to perform the gate-level optimizing, mapping, placing, and routing.

7.3 USES AND VARIATIONS OF C COMPILATION TO HARDWARE

Now that we have covered the technical aspects of compiling C to hardware, we will return to higher-level programming and system-level design.

7.3.1 Automatic HW/SW Partitioning

Once we have a common source language, here C, and compilation tools that can compile a program, or parts of it, to either the CPU or the reconfigurable fabric, the remaining problem is to partition the program between the

two resources. This partitioning can be performed manually, with the user adding annotations about where to run blocks of code (e.g., loops, procedures), automatically, with the compiler making all the decisions, or some combination of the two.

Even when partitioning is manual, the use of a common source language allows rapid exploration of the design space of different HW/SW mappings. The program can be written and debugged entirely on the CPU and the programmer need only modify the allocation directives to move code onto the hardware or to change which code is allocated to it. Profiling can help the user converge on a good split.

Nonetheless, in the purely manual case the program developed ends up tuned to a specific machine, with a specific amount of hardware, specific relative speeds for the RF and the CPU, and communication between the two. Ideally, we have a single source program to run on multiple hardware platforms with varying hardware and performance. An intermediate solution is for the directives to *suggest* which software blocks might be most profitable on the RF, then to allow the compiler, perhaps with runtime feedback, to decide which of the suggested set to actually run on the hardware based on performance benefits and capacity.

Ultimately, the compiler and runtime system should take full responsibility for determining the right code and granularity to move to the reconfigurable fabric. This is an active area of research and development. Chapter 26 discusses issues and techniques for hardware/software partitioning in more detail.

The Garp C compiler [5,7] provides an example of automatic partitioning. It starts by marking all loops as candidates for the reconfigurable fabric. Then, for each loop, it removes any paths from this candidate that include operations not supported on the array (removed paths are executed in software on the CPU). The compiler further trims the less taken paths in the loop until the remaining loop paths fit on the fabric capacity. Finally, it trims paths to improve performance. At this point, if any paths remain in the candidate loop, the compiler evaluates HW versus SW performance for the loop, considering the overhead costs for paths switching between HW and SW. If a loop is faster on the CPU, it is given a completely SW implementation. The Garp hardware supports fast configuration loads, and it caches configurations in the array, so there is a hard bound to the size of each loop but no limit on the number of accelerated loops.

For conventional FPGAs that do not support fast configuration swaps, it may be necessary to allocate all hardware logic at startup and keep them resident throughout operation. In these cases, the bound is on the total capacity of all hardware allocated to the RF, not just a single loop. The compiler may start with all feasible candidates, as in the Garp C compiler case, but then must select a subset that fits in the available capacity and maximizes performance.

7.3.2 Programmer Assistance

Useful code changes

As Section 7.2.4 shows, the compiler does many things to try to expose parallelism and optimize the implementation. However, discovering many of the

optimization opportunities requires very sophisticated analysis by the compiler, and sometimes it simply cannot prove that a particular optimization is always safe. Consequently, there are many ways a programmer might restructure or modify the application code to assist the compiler and achieve better performance on the target system. Some of these transformations have been studied to some degree in a research setting, but have not yet been fully automated in production compilers.

Loop interchange, reversal, and other transforms A loop nest can be altered in ways that still obey all required scalar and memory dependencies but that improve performance. For example, a compiler may automatically exploit memory accesses that are unit stride (A[0], A[1], A[2], ...) by streaming or prefetching. Even without explicit stream fetch support, unit stride accesses will improve cache locality, so the programmer should strive for them within the innermost loops. From one iteration to the next, loop interchange typically affects the loop-carried dependencies of the innermost loop; this impacts how effectively the block can be pipelined. If the programmer can structure the loop nest so that the innermost loop has no loop-carried dependencies, pipelining will be very effective. When the unit of HW implementation is an inner loop, another consideration is the overhead of switching between SW and HW execution. To reduce the relative cost of the overhead, it is best if possible to interchange the loops so that the innermost loops have high loop counts—as long as this does not adversely affect other aspects such as cache performance, unit stride, or loop-carried dependencies.

Loop fusion and fission Loop fusion is the combining of successive loops with identical bounds. This can remove memory accesses if the second loop loads values written by the first loop; instead, the value can be passed directly within the fused loop. The reverse, loop fission (splitting one loop into two), can also be useful when the original loop cannot fit in its entirety on the reconfigurable resources. Afterward, the two halves can each fit, but not at the same time, so temporary arrays may need to be introduced to store data produced in the first half and used in the second.

Local arrays When an array is local to a procedure and of fixed size, it is relatively easy for the compiler to do the "smart thing" and implement it using a memory block on the FPGA fabric. But if the program instead uses `malloc`'d or global arrays as temporaries, it is very challenging to safely convert them to local arrays. Thus, changing the code to use local arrays wherever possible can be very useful because on-FPGA memory blocks have much lower latency to/from the computation unit and can be accessed in parallel with each other.

Control structure Most compliers keep the loop, procedure, and block structure in the original code. As noted previously, common heuristics for hardware/software partitioning select loop bodies or procedures as candidates for hardware implementation. If the loop is too large, it may not be feasible on

the array. If the loop is too small, it might not make good use of the array's parallelism. The programmer can often assist the compiler by sizing and organizing loops, procedures, and blocks that make good candidates for hardware allocation.

Address indirection As noted in Section 7.2.3, whenever the address of a variable is taken, the compiler must make conservative assumptions about when the value will be updated, forcing additional sequentialization and increasing memory traffic. Consequently, address indirection and pass-by-reference should be used judiciously with the realization that it can inhibit compiler optimizations. Note that this unfortunate effect can also occur when a global scalar variable is visible beyond the file in which it is declared; with separate compilation, the compiler must assume that code in some other file takes the address of the variable and passes it back as a pointer. Therefore, declaring file-global variables as static helps as well.

Declaration of data sizes On CPUs there is often little advantage to using a narrow data word. Except for low-cost embedded systems, all processors have at least 32-bit words, with high-performance processors trending to 64 bit; even DSPs and embedded processors can typically assume CPUs with at least 16-bit words. Consequently, there is little incentive to software programmers to pay much attention to the actual range of data used. However, in fine-grained reconfigurable fabrics, such as field-programmable gate arrays (FPGAs), narrow data words can be implemented with less area and, sometimes, with less delay. As noted in Section 7.2.3, the compiler can make use of narrower type declarations (e.g., `short`, `char`) to reduce operator size.

Useful annotations

A programmer annotation gives the compiler a guarantee about a certain property of the program, which typically allows the compiler to make more aggressive optimizations; however, if the programmer is in error and the guarantee does not hold in all cases, incorrect program behavior may result. Some annotations can be expressed as assertions. If the assertion fails, the program will terminate, signaling the user (hopefully, the programmer) that the assertion was violated. The compiler knows that when execution continues past the assertion, certain properties must hold.

Annotations and assertions can be used as ways to communicate information to the compiler that it is not capable of inferring itself. In this way they may be an alternative to very advanced compiler analysis, or a complement when the analysis is simply intractable. Following are two examples of useful annotations:

- *Pointer independence:* declaring that a pair of pointers will never point to the same location, so that an ordering edge between accesses using those pointers can always be removed safely.
- *Absence of loop-carried memory dependences:* declaring that the memory operations in different iterations of the loop are always independent (to

different locations), which typically allows much greater overlap and greater performance when using pipelined scheduling.

Integrating operator-level modules

Even when writing C code for CPUs, the compiler does not always generate optimal machine code, and it is occasionally necessary to write assembly code for key routines. Similarly, when the C compiler does not provide the tight implementations of which the RF is capable, it may be necessary to provide a direct hardware implementation. Here, the "assembly" may be a VHDL (Chapter 6) implementation of a function or a piece of dataflow. As in the assembly language case, the developer can start with a pure C program profile, the code, and then judiciously spend his customization effort on the code's most performance-critical regions.

It is fairly easy to integrate a custom operation into the flow we have described. The designer simply needs to create the module via HDL or schematic capture, and tell the compiler the latency, in cycles, of the design. The operation can be accessed from C source code using function call syntax, instantiated, and scheduled in parallel with other "native" C operations in the hyperblock. For example, in this code snippet:

```
x = bitreverse(a);
y = a ^ b;
z = x + y;
```

the `bitreverse` module would have one cycle latency and could be scheduled in parallel with the XOR (^) module.

The power of this approach is greatly increased with a *module generator*. In this case, the HDL module is not just copied from a library; instead, it is dynamically generated by the compiler. This allows constant arguments to the module instantiation to specialize it, for example,

```
X = bit_reverse_range(a,8,15);
```

which will generate a module that will reverse the bits of `a` from bit 8 to bit 15 to produce `x`. A detailed interface between compiler and dynamic module generator is described in work by Koch [10] (see also Chapter 15).

It is useful to always have a functionally equivalent software implementation of each custom operation in order to enable testing of the overall application in a pure software environment. This is required, for example, when adding hand-designed HDL modules in the SRC Computers compiler [14].

Integrating large blocks

Another method for integrating a hand-designed circuit with an otherwise C-compiled program is to treat it as its own hyperblock subcircuit within the compiler, allowing it to manage its own sequencing. The HDL implementation of the custom block in this case receives a `start` control bit, like any other hyperblock, and must send a finish control bit when done. This allows the designer to incorporate custom blocks that have variable latency (e.g., an iterative divider or

a greatest-common-divisor computation). The programmer could use function call syntax to instantiate this larger block as well, but, the compiler would prevent the function from being merged with other blocks into a larger hyperblock.

7.4 SUMMARY

After a decade of research, C compilation for reconfigurable computers is now commercially available in many forms (e.g., SRC Computers [14] and Lau and colleagues [11]). While today's commercial compilers cannot generally compile arbitrary ISO C code or take arbitrary C code and expect to fully extract the performance of the reconfigurable fabric, they have closed the gap so that nontrivial code acceleration is possible with minor programmer effort. A developer can use the C compiler to rapidly get applications running on a suitable reconfigurable platform. C code developed or tuned with an understanding of the reconfigurable platform and the capabilities of the compiler can achieve higher performance. Although today's C compilers do not free the reconfigurable developer from understanding good application and system architectures, they can allow her to focus her efforts.

C compilation and optimization remain an active area of research, and we expect to see continuing improvements over time. Many opportunities exist for innovative research on aggressive optimization techniques and development of more automated optimizing complier flows.

References

[1] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers, Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[2] M. Budiu, S. Copen Goldstein. Optimizing memory accesses for spatial computation. *International ACM/IEEE Symposium on Code Generation and Optimization*, March 2003.

[3] M. Budiu, M. Sakr, K. Walker, S. Copen Goldstein. Bit value inference: Detecting and exploiting narrow bit-width computations. *European Conference on Parallel Processing*, Springer-Verlag, 2000.

[4] M. Budiu. *Spatial Computation*, Ph.D. thesis, Carnegie-Mellon University, December 2003 (technical report CMU-CS-03-217).

[5] T. J. Callahan. *Automatic Compilation of C for Hybrid Reconfigurable Architectures*, Ph.D. thesis, University of California, Berkeley, December 2002.

[6] T. J. Callahan, P. Chong, A. DeHon, J. Wawrzynek. Rapid module mapping and placement for FPGAs. *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 1998.

[7] T. Callahan, J. Hauser, J. Wawrzynek. The Garp architecture and C compiler. *IEEE Computer* 33(4), April 2000.

[8] T. Callahan, J. Wawrzynek. Adapting software pipelining for reconfigurable computing. *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2000.

[9] P. Harbison, G. L. Steele. *C, A Reference Manual*, 4th ed. Prentice-Hall, 1995.

[10] A. Koch. Compilation for adaptive computing systems using complex parameterized hardware objects. *Journal of Supercomputing* 21(2), 2002.

[11] D. Lau, O Pritchard, P. Molson. Automated generation of hardware accelerators with direct memory access from ANSI/ISO standard C functions. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2006.

[12] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 1992.

[13] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[14] SRC Computers. *SRC Carte C Programming Environment v2.2 Guide*, Colorado Springs, 2007.

CHAPTER 8

PROGRAMMING STREAMING FPGA APPLICATIONS USING BLOCK DIAGRAMS IN SIMULINK

Brian C. Richards, Chen Chang, John Wawrzynek, Robert W. Brodersen
Department of Electrical Engineering and Computer Science
University of California–Berkeley

Although a system designer can use hardware description languages, such as VHDL (Chapter 6) and Verilog to program field-programmable gate arrays (FPGAs), the algorithm developer typically uses higher-level descriptions to refine an algorithm. As a result, an algorithm described in a language such as Matlab or C is frequently reentered by hand by the system designer, after which the two descriptions must be verified and refined manually. This can be time consuming.

To avoid reentering a design when translating from a high-level simulation language to HDL, the algorithm developer can describe a system from the beginning using block diagrams in Matlab Simulink [1]. Other block diagram environments can be used in a similar way, but the tight integration of Simulink with the widely used Matlab simulation environment allows developers to use familiar data analysis tools to study the resulting designs. With Simulink, a single design description can be prepared by the algorithm developer and refined jointly with the system architect using a common design environment.

The single design entry is enabled by a library of Simulink operator primitives that have a direct mapping to HDL, using matching Simulink and HDL models that are cycle accurate and bit accurate between both domains. Examples and compilation environments include System Generator from Xilinx [2], Synplify DSP from Synplicity [3], and the HDL Coder from The Mathworks [1]. Using such a library, nearly any synchronous multirate system can be described, with high confidence that the result can be mapped to an FPGA given adequate resources.

In this chapter, a high-performance image-processing system is described using Simulink and mapped to an FPGA-based platform using a design flow built around the Xilinx System Generator tools. The system implements edge detection in real time on a digitized video stream and produces a corresponding video stream labeling the edges. The edges can then be viewed on a high-resolution monitor. This design demonstrates how to describe a high-performance parallel

datapath, implement control subsystems, and interface to external devices, including embedded processors.

8.1 DESIGNING HIGH-PERFORMANCE DATAPATHS USING STREAM-BASED OPERATORS

Within Simulink we employ a Synchronous Dataflow computational model (SDF), described in the Synchronous dataflow subsection of Section 5.1.3. Each operator is executed once per clock cycle, consuming input values and producing new output values once per clock tick. This discipline is well suited for stream-based design, encouraging both the algorithm designer and the system architect to describe efficient datapaths with minimal idle operations.

Clock signals and corresponding clock enable signals do not appear in the Simulink block diagrams using the System Generator libraries, but are automatically generated when an FPGA design is compiled. To support multirate systems, the System Generator library includes up-sample and down-sample blocks to mark the boundaries of different clock domains. When compiled to an FPGA, clock enable signals for each clock domain are automatically generated.

All System Generator components offer compile time parameters, allowing the designer to control data types and refine the behavior of the block. Hierarchical blocks, or *subsystems* in Simulink, can also have user-defined parameters, called *mask parameters*. These can be included in block property expressions within that subsystem to provide a means of generating a variety of behaviors from a single Simulink description. Typical mask parameters include data type and precision specification and block latency to control pipeline stage insertion. For more advanced library development efforts, the mask parameters can be used by a Matlab program to create a custom schematic at compile time.

The System Generator library supports fixed-point or Boolean data types for mapping to FPGAs. Fixed-point data types include signed and unsigned values, with bit width and decimal point location as parameters. In most cases, the output data types are inferred automatically at compile time, although many blocks offer parameters to define them explicitly.

Pipeline operators are explicitly placed into a design either by inserting delay blocks or by defining a delay parameter in selected functional blocks. Although the designer is responsible for balancing pipeline operators, libraries of high-level components have been developed and reused to hide pipeline-balancing details from the algorithm developer.

The Simulink approach allows us to describe highly concurrent SDF systems where many operators—perhaps the entire dataflow path—can operate simultaneously. With modern FPGAs, it is possible to implement these systems with thousands of simultaneous operators running at the system clock rate with little or no control logic, allowing complex, high-performance algorithms to be implemented.

8.2 AN IMAGE-PROCESSING DESIGN DRIVER

The goal of the edge detection design driver is to generate a binary bit mask from a video source operating at up to a 200 MHz pixel rate, identifying where likely edges are in an image. The raw color video is read from a neighboring FPGA over a parallel link, and the image intensity is then calculated, after which two 3×3 convolutional Sobel operator filters identify horizontal and vertical edges; the sum of their absolute values indicates the relative strength of a feature edge in an image. A runtime programmable gain (variable multiplier) followed by an adjustable threshold maps the resulting pixel stream to binary levels to indicate if a given pixel is labeled as an edge of a visible feature. The resulting video mask is then optionally mixed with the original color image and displayed on a monitor.

Before designing the datapaths in the edge detection system, the data and control specification for the video stream sources and sinks must be defined. By convention, stream-based architectures are implemented by pairing data samples with corresponding control tags and maintaining this pairing through the architecture. For this example, the video datastreams may have varying data types as the signals are processed whereas the control tags are synchronization signals that track the pipeline delays in the video stream. The input video stream and output display stream represent color pixel data using 16 bits—5 bits for red, 6 bits for green, and 5 bits for blue unsigned pixel intensity values. Intermediate values might represent video data as 8-bit grayscale intensity values or as 1-bit threshold detection mask values.

As the datastreams flow through the signal-processing datapath, the operators execute at a constant 100 MHz sample rate, with varying pipeline delays through the system. The data, however, may arrive at less than 100 MHz, requiring a corresponding `enable` signal (see the discussion in Data presence subsection of Section 5.2.1) to tag valid data. Additionally, `hsync`, `vsync`, and `msync` signals are defined to be true for the first pixel of each row, frame, and movie sequence, respectively, allowing a large variety of video stream formats to be supported by the same design.

Once a streaming format has been specified, library components can be developed that forward a video stream through a variety of operators to create higher-level functions while maintaining valid, pipeline-delayed synchronization signals. For blocks with a pipeline latency that is determined by mask parameters, the synchronization signals must also be delayed based on the mask parameters so that the resulting synchronization signals match the processed datastream.

8.2.1 Converting RGB Video to Grayscale

The first step in this example is to generate a grayscale video stream from the RGB input data. The data is converted to intensity using the NTSC RGB-to-Y matrix:

$$Y = 0.3 \times \text{red} + 0.59 \times \text{green} + 0.11 \times \text{blue}$$

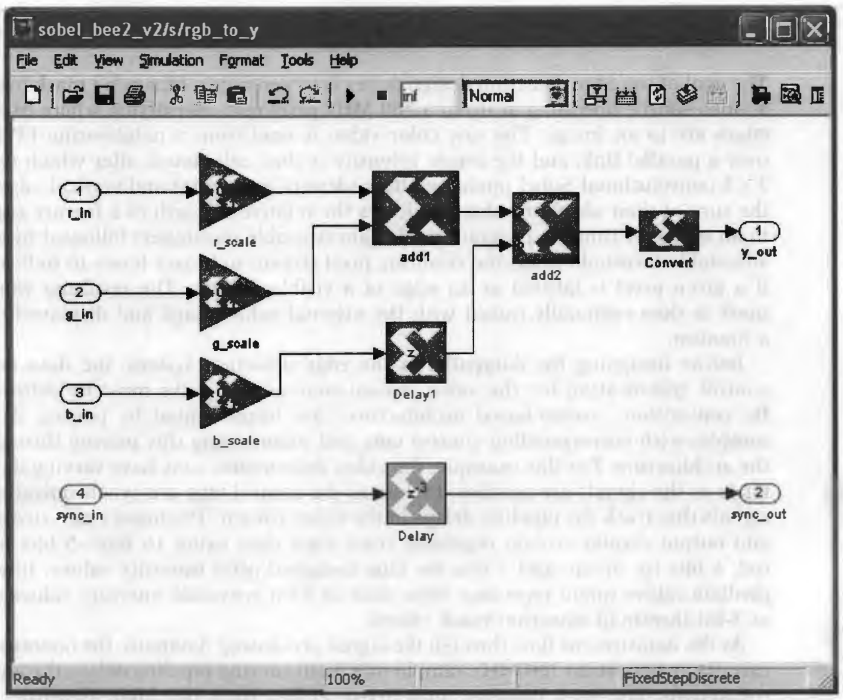


FIGURE 8.1 ■ An RGB-to-*Y* (intensity) Simulink diagram.

This formula is implemented explicitly as a block diagram, shown in Figure 8.1, using constant gain blocks followed by adders. The constant multiplication values are defined as floating-point values and are converted to fixed point according to mask parameters in the gain model. This allows the precision of the multiplication to be defined separately from the gain, leaving the synthesis tools to choose an implementation. The scaled results are then summed with an explicit adder tree.

Note that if the first adder introduces a latency of `adder_delay` clock cycles, the `b` input to the second adder, `add2`, must also be delayed by `adder_delay` cycles to maintain the cycle alignment of the RGB data. Both the `Delay1` block and the `add1` block have a subsystem mask parameter defining the delay that the block will introduce, provided by the mask parameter dialog as shown in Figure 8.2. Similarly, the synchronization signals must be delayed by three cycles corresponding to one cycle for the gain blocks, one cycle for the first adder,

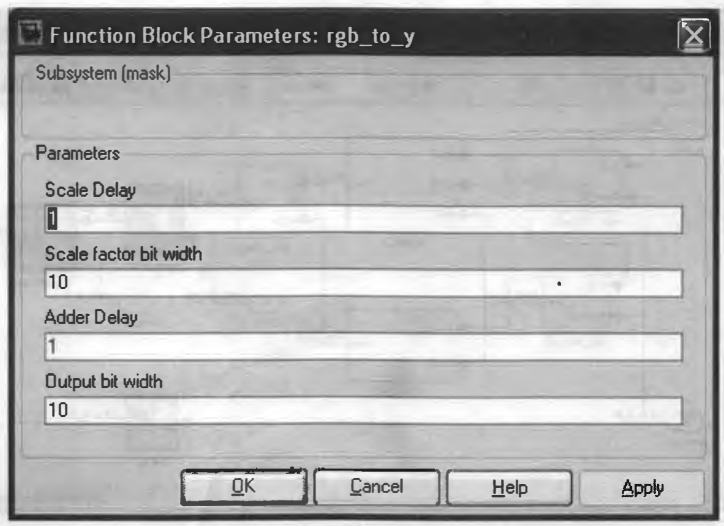


FIGURE 8.2 ■ A dialog describing mask parameters for the `rgb_to_y` block.

and one cycle for the second adder. By designing subsystems with configurable delays and data precision parameters, library components can be developed to encourage reuse of design elements.

8.2.2 Two-dimensional Video Filtering

The next major block following the RGB-to-grayscale conversion is the edge detection filter itself (Figure 8.3), consisting of two pixel row delay lines, two 3×3 kernels, and a simplified magnitude detector. The delay lines store the two rows of pixels preceding the current row of video data, providing three streams of vertically aligned pixels that are connected to the two 3×3 filters—the first one detecting horizontal edges and the second detecting vertical edges. These filters produce two signed fixed-point streams of pixel values, approximating the edge gradients in the source video image.

On every clock cycle, two 3×3 convolution kernels must be calculated, requiring several parallel operators. The operators implement the following convolution kernels:

Sobel X Gradient:

−1	0	+1
−2	0	+2
−1	0	+1

Sobel Y Gradient:

+1	+2	+1
0	0	0
−1	−2	−1

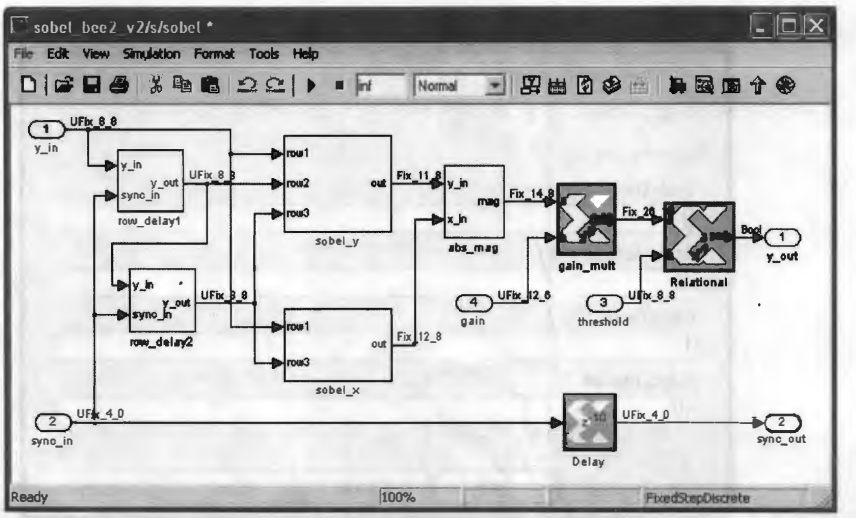


FIGURE 8.3 ■ The Sobel edge detection filter, processing an 8-bit video datastream to produce a stream of Boolean values indicating edges in the image.

To support arbitrary kernels, the designer can choose to implement the Sobel operators using constant multiplier or gain blocks followed by a tree of adders. For this example, the subcircuits for the x- and y-gradient operators are hand-optimized so that the nonzero multipliers for both convolution kernels are implemented with a single hardwired shift operation using a power-of-2 scale block. The results are then summed explicitly, using a tree of add or subtract operators, as shown in Figures 8.4 and 8.5.

Note that the interconnect in Figures 8.4 and 8.5 is shown with the data types displayed. For the most part, these are assigned automatically, with the input data types propagated and the output data types and bit widths inferred to avoid overflow or underflow of signed and unsigned data types. The bit widths can be coerced to different data types and widths using casting or reinterpret blocks, and by selecting saturation, truncation, and wraparound options available to several of the operator blocks. The designer must exercise care to verify that such adjustments to a design do not change the behavior of the algorithm.

Through these Simulink features a high-level algorithm designer can directly explore the impact of such data type manipulation on a particular algorithm.

Once the horizontal and vertical intensity gradients are calculated for the neighborhood around a given pixel, the likelihood that the pixel is near the boundary of a feature can be calculated. To label a pixel as a likely edge of a feature in the image, the magnitude of the gradients is approximated and the

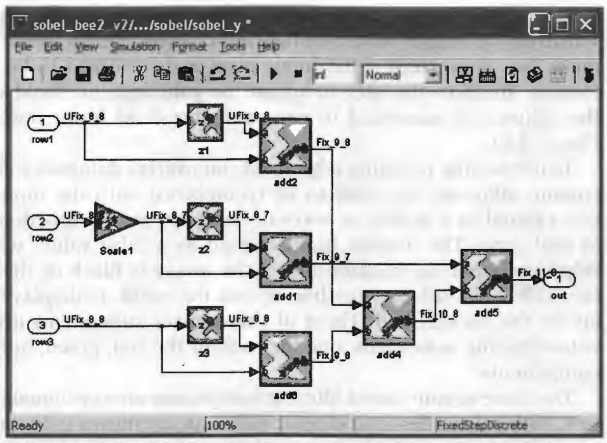


FIGURE 8.4 ■ The `sobel_y` block for estimating the horizontal gradient in the source image.

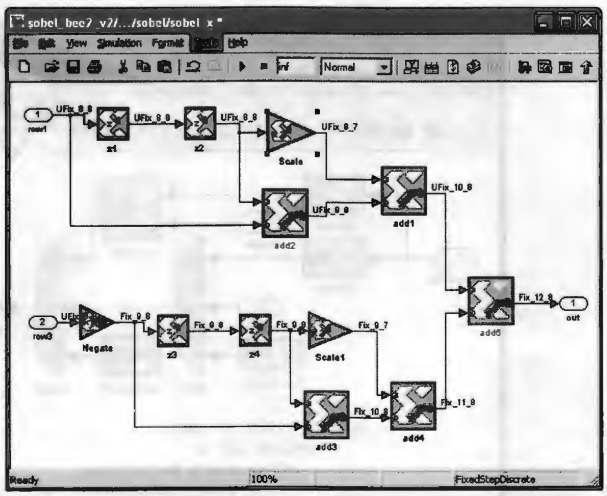


FIGURE 8.5 ■ The `sobel_x` block for estimating the vertical gradient in the source image.

resulting nonnegative value is scaled and compared to a given threshold. The magnitude is approximated by summing the absolute values of the horizontal and vertical edge gradients, which, although simpler than the exact magnitude calculation, gives a result adequate for our applications.

A multiplier and a comparator follow the magnitude function to adjust the sensitivity to image noise and lighting changes, respectively, resulting in a 1-bit mask that is nonzero if the input pixel is determined to be near the edge of a feature. To allow the user to adjust the gain and threshold values interactively, the values are connected to gain and threshold input ports on the filter (see Figure 8.6).

To display the resulting edge mask, an overlay datapath follows the edge mask stream, allowing the mask to be recombined with the input RGB (red, green, blue) signal in a variety of ways to demonstrate the functionality of the system in real time. The overlay input is read as a 2-bit value, where the LSB 0 bit selects whether the background of the image is black or the original RGB, and the LSB 1 bit selects whether or not the mask is displayed as a white overlay on the background. Three of these mixer subsystems are used in the main video-filtering subsystem, one for each of the red, green, and blue video source components.

The three stream-based filtering subsystems are combined into a single subsystem, with color video in and color video out, as shown in Figure 8.7. Note that the color data fed straight through to the red, green, and blue mixers is delayed. The delay, 13 clock cycles in this case, corresponds to the pipeline delay through both

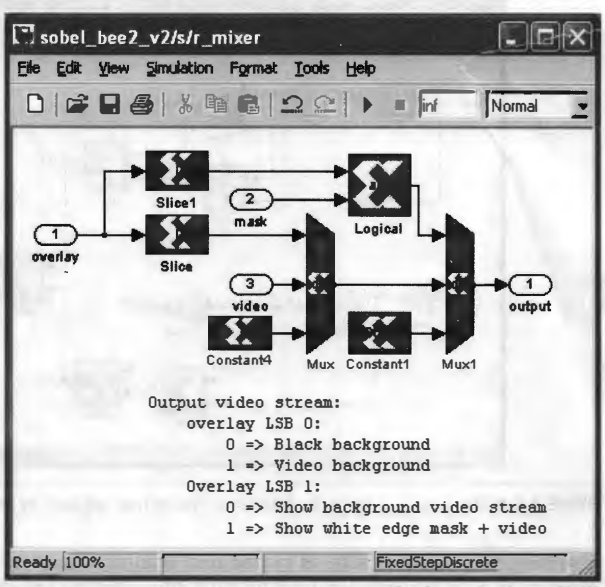


FIGURE 8.6 ■ One of three video mixers for choosing displays of the filtered results.

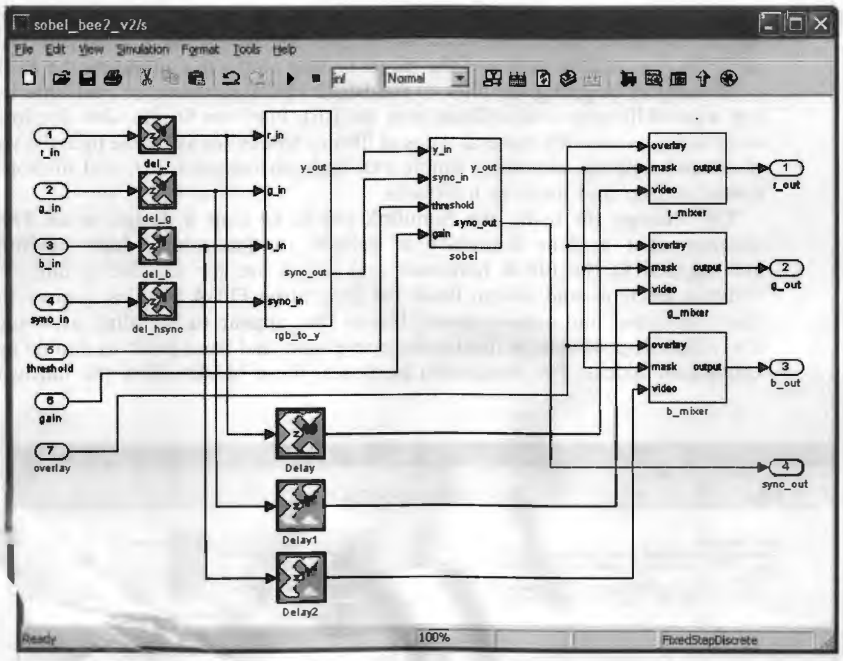


FIGURE 8.7 ■ The main filtering subsystem, with RGB-to-*Y*, Sobel, and mixer blocks.

the `rgb_to_y` block and the Sobel edge detection filter itself. This is to ensure that the background original image data is aligned with the corresponding pixel results from the filter. The sync signals are also delayed, but this is propagated through the filtering blocks and does not require additional delays.

8.2.3 Mapping the Video Filter to the BEE2 FPGA Platform

Our design, up to this point, is platform independent—any Xilinx component supported by the System Generator commercial design flow can be targeted. The next step is to map the design to the BEE2 platform—a multiple-FPGA design, developed at UC Berkeley [4], that contains memory to store a stream of video data and an HDMI interface to output that data to a high-resolution monitor.

For the Sobel edge detection design, some ports are for video datastreams and others are for control over runtime parameters. The three user-controllable inputs to the filtering subsystem, threshold, gain, and overlay are connected to external input ports, for connection to the top-level testbench. The filter,

included as a subsystem of this testbench design, is shown in Figures 8.8 and 8.9. So far, the library primitives used in the filter are independent of both the type of FPGA that will be used and the target testing platform containing the FPGA.

To support targeting the filter to the BEE2 FPGA platform for real-time testing, a set of libraries and utilities from the BEE Platform Studio, also developed at Berkeley, is used [5]. Several types of library blocks are available to assist with platform mapping, including simple I/O, high-performance I/O, and microprocessor register and memory interfaces.

The strategy for using the Simulink blocks to map a design to an FPGA assumes that a clear boundary is defined to determine which operators are mapped to the FPGA hardware and which are for simulation only. The commercial tools and design flows for generating FPGA bit files assume that there are input and output library blocks that appear to Simulink as, respectively, double-precision to fixed-point conversion and fixed-point to double type conversion blocks. For simulation purposes, these blocks allow the hardware

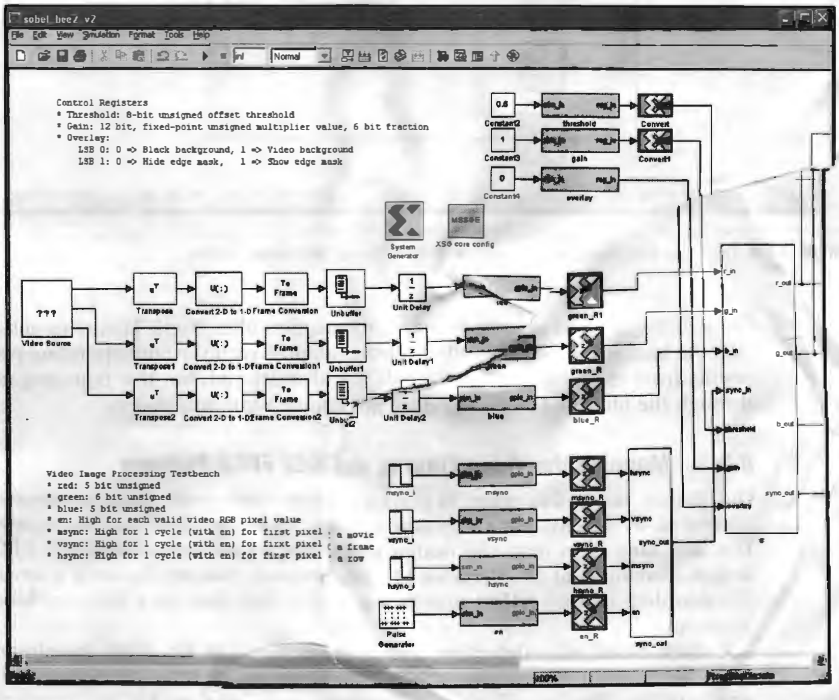


FIGURE 8.8 ■ The top-level video testbench, with input, microprocessor register, and configuration blocks.

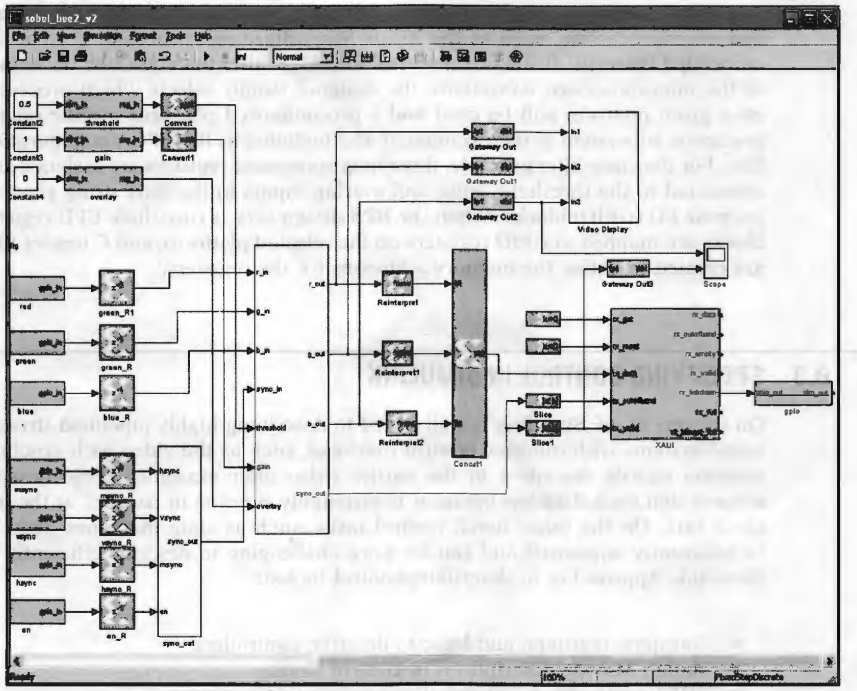


FIGURE 8.9 ■ The output section of the top-level testbench, with a 10G XAUI interface block.

description to be simulated with a software testbench to verify basic functionality before mapping the design to hardware. They also allow the designer to assign the FPGA pin locations for the final configuration files.

The BEE Platform Studio (BPS) [5] provides additional I/O blocks that allow the designer to select pin locations symbolically, choosing pins that are hardwired to other FPGAs, LEDs, and external connections on the platform. The designer is only required to select a platform by setting BPS block parameters, and does not need to keep track of I/O pin locations. This feature allows the designer to experiment with architectural tradeoffs without becoming a hardware expert.

In addition to the basic I/O abstractions, the BPS allows high-performance or analog I/O devices to be designed into a system using high-level abstractions. For the video-testing example, a 10 Gbit XAUI I/O block is used to output the color video stream to platform-specific external interfaces. The designer selects the port to be used on the actual platform from a pulldown menu of available names, hiding most implementation details.

A third category of platform-specific I/O enables communication with embedded microprocessors, such as the Xilinx MicroBlaze soft processor core or the embedded PowerPC available on several FPGAs. Rather than describe the details of the microprocessor subsystem, the designer simply selects which processor on a given platform will be used and a preconfigured platform-specific microprocessor subsystem is then generated and included in the FPGA configuration files. For the video filter example, three microprocessor registers are assigned and connected to the threshold, gain, and overlap inputs to the filter using general-purpose I/O (GPIO) blocks. When the BPS design flow is run, these CPU register blocks are mapped to GPIO registers on the selected platform, and C header files are created to define the memory addresses for the registers.

8.3 SPECIFYING CONTROL IN SIMULINK

On the one hand, Simulink is well suited to describing highly pipelined stream-based systems with minimal control overhead, such as the video with synchronization signals described in the earlier video filter example. These designs assume that each dataflow operator is essentially running in parallel, at the full clock rate. On the other hand, control tasks, such as state machines, tend to be inherently sequential and can be more challenging to describe efficiently in Simulink. Approaches to describing control include:

- Counters, registers, and logic to describe controllers
- Matlab M-code descriptions of control blocks
- VHDL or Verilog hand-coded or compiled descriptions
- Embedded microprocessors

To explore the design of control along with a stream-based datapath, consider the implementation of a synchronous delay line based on a single-port memory. The approach described here is to alternate between writing two data samples and reading two data samples on consecutive clock cycles. A simpler design could be implemented using dual-port memory on an FPGA, but the one we are using allows custom SOC designs to use higher-density single-port memory blocks.

8.3.1 Explicit Controller Design with Simulink Blocks

The complete synchronous delay line is shown in Figure 8.10. The control in this case is designed around a counter block, where the least significant bit selects between the two words read or written from the memory on a given cycle and the upper counter bits determine the memory address. In addition to the counter, control-related blocks include *slice* blocks to select bit fields and Boolean *logic* blocks. For this design, the block diagram is effective for describing control, but minor changes to the controller can require substantial redesign.

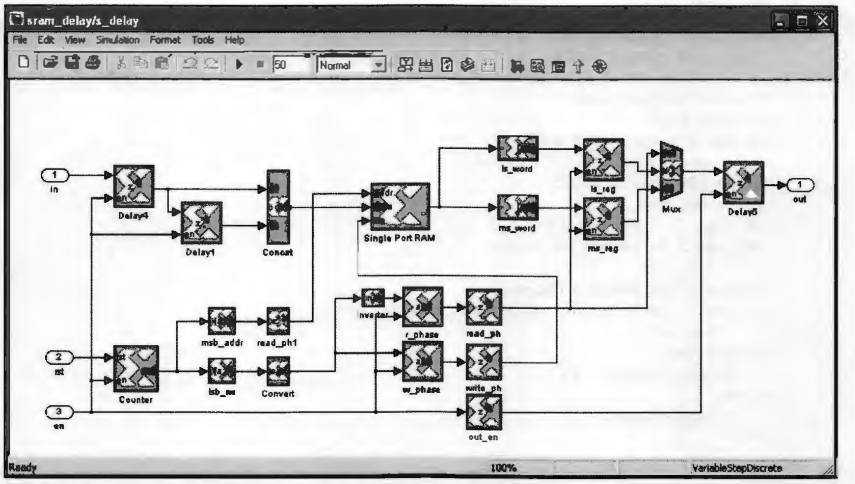


FIGURE 8.10 ■ A simple datapath with associated explicit control.

8.3.2 Controller Design Using the Matlab M Language

For a more symbolic description of the synchronous delay line controller, the designer can use the Matlab "M" language to define the behavior of a block, with the same controller described previously written as a Matlab function. Consider the code in Listing 8.1 that is saved in the file `sram_delay_cntl.m`.

Listing 8.1 ■ The delay line controller described with the Matlab function `sram_delay_cntl.m`.

```
function [addr, we, sel] = sram_delay_cntl(rst, en, counter_bits, counter_max)
% sram_delay_cntl -- MCode implementation block.
% Author: Brian Richards, 11/16/2005, U. C. Berkeley
%
% The following Function Parameter Bindings should be declared in
% the MCode block Parameters (sample integer values are given):
%   {'counter_bits', 9, 'counter_max', 5}

% Define all registers as persistent variables.
persistent count,
    count = xl_state(0, {xlUnsigned, counter_bits, 0});
persistent addr_reg,
    addr_reg = xl_state(0, {xlUnsigned, counter_bits-1, 0});
persistent we_reg,     we_reg    = xl_state(0, {xlBoolean});
persistent sel_reg_1, sel_reg_1 = xl_state(0, {xlBoolean});
persistent sel_reg_2, sel_reg_2 = xl_state(0, {xlBoolean});

% Delay the counter output, and split the lsb from
% the upper bits.
```

```
addr = addr_reg;
addr_reg = xl_slice(count, counter_bits-1, 1);
count_lsb = xfix({xlBoolean}, xl_slice(count, 0, 0));

% Write-enable logic
we = we_reg;
we_reg = count_lsb & en;

% MSB-LSB select logic
sel = sel_reg_2;
sel_reg_2 = sel_reg_1;
sel_reg_1 = ~count_lsb & en;

% Update the address counter:
if (rst | (en & (count == counter_max)))
    count = 0;
elseif (en)
    count = count + 1;
else
    count = count;
end
```

To add the preceding controller to a design, the Xilinx M-code block can be dragged from the Simulink library browser and added to the subsystem. A dialog box then asks the designer to select the file containing the M source code, and the block `sram_delay_cntl` is automatically created and added to the system (see Figure 8.11).

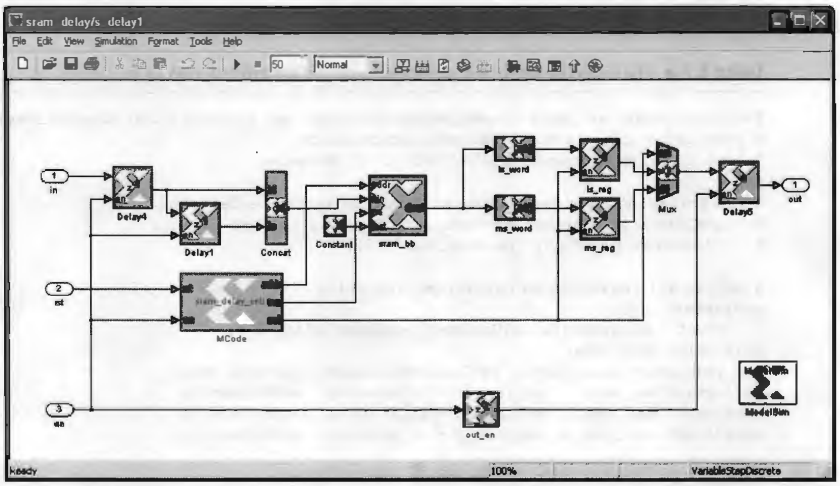


FIGURE 8.11 ■ A simple datapath using a controller described in Matlab code.

There are several advantages to using the M-code description compared to its explicit block diagram equivalent. First, large, complex state machines can be described and documented efficiently using the sequential M language. Second, the resulting design will typically run faster in Simulink because many fine-grained blocks are replaced by a single block. Third, the design is mapped to an FPGA by generating an equivalent VHDL RTL description and synthesizing the resulting controller; the synthesis tools can produce different results depending on power, area, and speed constraints, and can optimize for different FPGA families.

8.3.3 Controller Design Using VHDL or Verilog

As in the M language approach just described, a controller can also be described with a *black box* containing VHDL or Verilog source code. This approach can be used for both control and datapath subsystems and has the benefit of allowing IP to be included in a Simulink design.

The VHDL or Verilog subsystems must be written according to design conventions to ensure that the subsystem can be mapped to hardware. Clocks and enables, for example, do not appear on the generated Simulink block, but must be defined in pairs (e.g., `clk_sg`, `ce_sg`) for each implied data rate in the system. Simulink designs that use these VHDL or Verilog subsystems can be verified by cosimulation between Simulink and an external HDL simulator, such as Modelsim [6]. Ultimately, the same description can be mapped to hardware, assuming that the hardware description is synthesizable.

8.3.4 Controller Design Using Embedded Microprocessors

The most elaborate controller for an FPGA is the embedded microprocessor. In this case, control can be defined by running compiled or interpreted programs on the microprocessor. On the BEE2 platform, a tiny shell can be used interactively to control datapath settings, or a custom C-based program can be built using automatically generated header files to symbolically reference hardware devices.

A controller implemented using an embedded microprocessor is often much slower than the associated datapath hardware, perhaps taking several clock cycles to change control parameters. This is useful for adjusting parameters that do not change frequently, such as threshold, gain, and overlay in the Sobel filter. The BEE Platform Studio design flow uses the Xilinx Embedded Development Kit (EDK) to generate a controller running a command line shell, which allows the user to read and modify configuration registers and memory blocks within the FPGA design. Depending on the platform, this controller can be accessed via a serial port, a network connection, or another interface port.

The same embedded controller can also serve as a source or sink for low-bandwidth datastreams. An example of a user-friendly interface to such a source or sink is a set of Linux 4.2 kernel extensions developed as part of the BEE operating system, BORPH [7]. BORPH defines the notion of a hardware process, where a bit file and associated interface information is encapsulated in an

executable `.bof` file. When launched from the Linux command line, a software process is started that programs and then communicates with the embedded processor on a selected FPGA. To the end user, hardware sources and sinks in Simulink are mapped to Linux files or pipes, including standard input and standard output. These file interfaces can then be accessed as software streams to read from or write to a stream-based FPGA design for debugging purposes or for applications with low-bandwidth continuous datastreams.

8.4 COMPONENT REUSE: LIBRARIES OF SIMPLE AND COMPLEX SUBSYSTEMS

In the previous sections, low-level primitives were described for implementing simple datapath and control subsystems and mapping them to FPGAs. To make this methodology attractive to the algorithm developer and system architect, all of these capabilities are combined to create reusable library components, which can be parameterized for a variety of applications; many of them have been tested in a variety of applications.

8.4.1 Signal-processing Primitives

One example of a rich library developed for the BPS is the Astronomy library, which was codeveloped by UC Berkeley and the Space Sciences Laboratory [8,9] for use in a variety of high-performance radio astronomy applications. In its simplest form, this library comprises a variety of complex-valued operators based on Xilinx System Generator real-valued primitives. These blocks are implemented as Simulink subsystems with optional parameters defining latency or data type constraints.

8.4.2 Tiled Subsystems

To enable the development of more sophisticated library components, Simulink supports the use of Matlab M language programs to create or modify the schematic within a subsystem based on parameters passed to the block. With the Simulink Mask Editor, initialization code can be added to a subsystem to place other Simulink blocks and to add interconnect to define a broad range of implementations for a single library component.

Figure 8.12 illustrates an example of a tiled cell, the `biplex_core` FFT block, which accepts several implementation parameters. The first parameters define the size and precision of the FFT operator, followed by the quantization behavior (truncation or rounding) and the overflow behavior of adders (saturation or wrapping). The pipeline latencies of addition and multiplication operators are also user selectable within the subsystem.

Automatically tiled library components can conditionally use different subsystems, and can have multiple tiling dimensions. An alternative to the stream-based `biplex_core` block shown in Figure 8.13, a parallel FFT implementation,

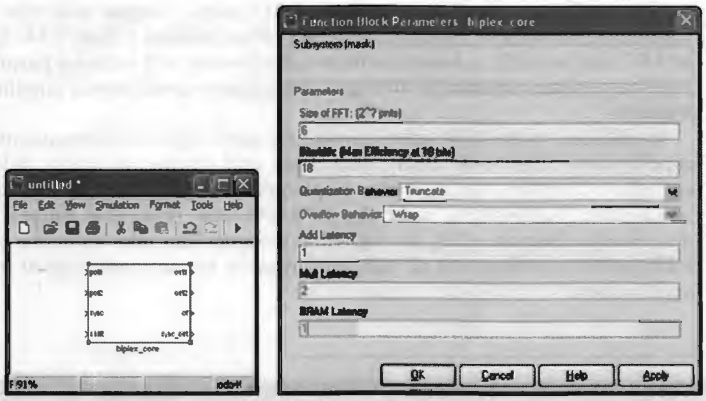


FIGURE 8.12 ■ The `biplex_core` dual-channel FFT block (a), with the parameter dialog box (b).

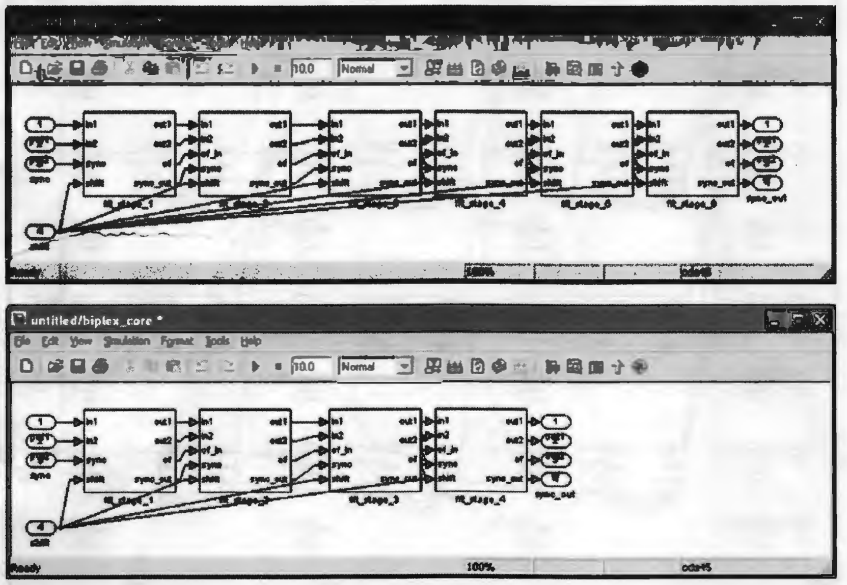


FIGURE 8.13 ■ Two versions of the model schematic for the `biplex_core` library component, with the size of the FFT set to 6 (2^6) and 4 (2^4). The schematic changes dynamically as the parameter is adjusted.

is also available, where the number of I/O ports changes with the FFT size parameter. An 8-input, 8-output version is illustrated in Figure 8.14. The parallel FFT tiles butterfly subsystems in two dimensions and includes parameterized pipeline registers so that the designer can explore speed versus pipeline latency tradeoffs.

In addition to the FFT, other commonly used high-level components include a poly-phase filter bank (PFB), data delay and reordering blocks, adder trees, correlator functions, and FIR filter implementations. Combining these platform-independent subsystems with the BPS I/O and processor interface library described in Section 8.2.3, an algorithm designer can take an active roll in the architectural development of high-performance stream-based signal-processing applications.

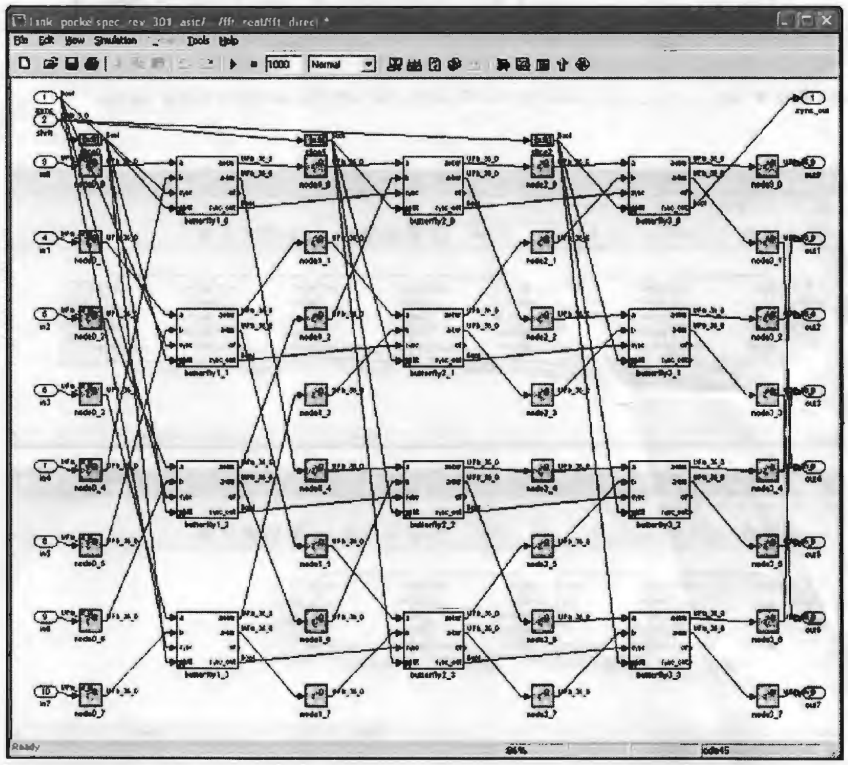


FIGURE 8.14 ■ An automatically generated 8-channel parallel FFT from the `fft_direct` library component.

8.5 SUMMARY

This chapter described the use of Simulink as a common design framework for both algorithm and architecture development, with an automated path to program FPGA platforms. This capability, combined with a rich library of high-performance parameterized stream-based DSP components, allows new applications to be developed and tested quickly.

The real-time Sobel video edge detection described in this chapter runs on the BEE2 platform, shown in Figure 8.15, which has a dedicated LCD monitor

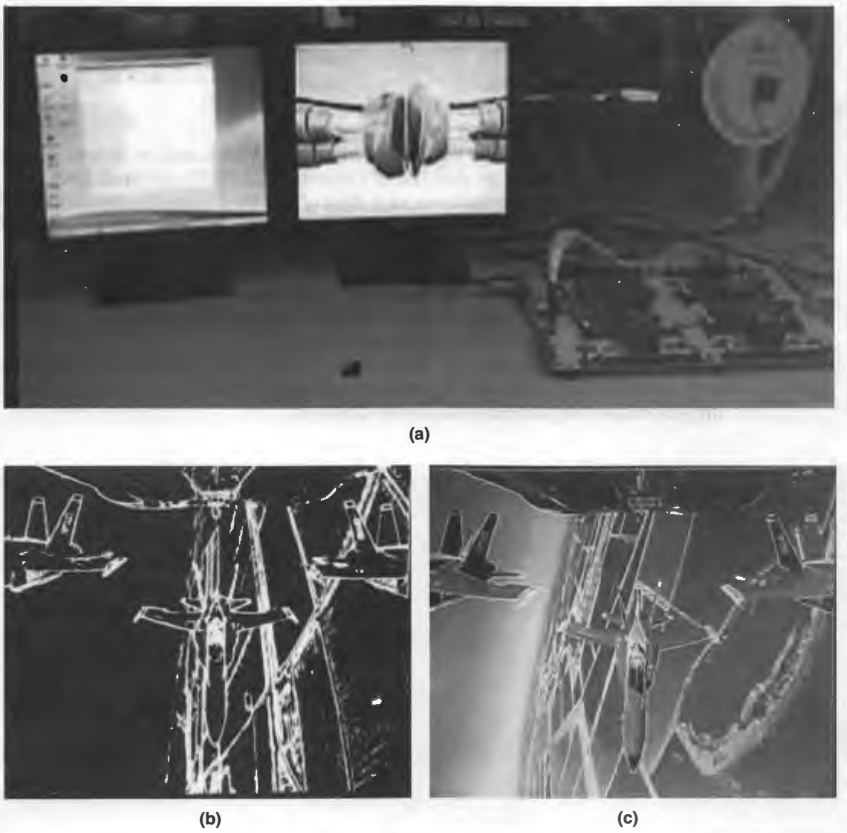


FIGURE 8.15 ■ (a) The Sobel edge detection filter running on the BEE2, showing the BEE2 console and video output on two LCD displays, with (b, c) two examples of edge detection results based on interactive user configuration from the console.

connected to it. Two filtered video samples are shown, with edges displayed with and without the original source color video image.

For more information on the BPS and related software, visit *http://bee2.eecs.berkeley.edu*, and for examples of high-performance stream-based library components, see the Casper Project [9].

Acknowledgments This work was funded in part by C2S2, the MARCO Focus Center for Circuit and System Solutions, under MARCO contract 2003-CT-888, and by Berkeley Wireless Research Center (BWRC) member companies (*bwrc.eecs.berkeley.edu*). The BEE Platform Studio development was done jointly with the Casper group at the Space Sciences Laboratory (*ssl.berkeley.edu/casper*).

References

[1] *http://www.mathworks.com.*
[2] *http://www.xilinx.com.*
[3] *http://www.synplicity.com.*
[4] C. Chang, J. Wawrzynek, R. W. Brodersen. BEE2: A high-end reconfigurable computing system. *IEEE Design and Test of Computers* 22(2), March/April 2005.
[5] C. Chang. *Design and Applications of a Reconfigurable Computing System for High Performance Digital Signal Processing*, Ph.D. thesis, University of California, Berkeley, 2005.
[6] *http://www.mentor.com.*
[7] K. Camera, H. K.-H. So, R. W. Brodersen. An integrated debugging environment for reprogrammble hardware systems. *Sixth International Symposium on Automated and Analysis-Driven Debugging*, September, 2005.
[8] A. Parsons et al. PetaOp/Second FPGA signal processing for SETI and radio astronomy. *Asilomar Conference on Signals, Systems, and Computers*, November 2006.
[9] *http://casper.berkeley.edu/papers/asilomar_2006.pdf.*

CHAPTER 9

STREAM COMPUTATIONS ORGANIZED FOR RECONFIGURABLE EXECUTION

André DeHon
Department of Electrical and Systems Engineering
University of Pennsylvania
Yury Markovskiy, Eylon Caspi, Michael Chu,
Randy Huang, Stylianos Perissakis, Laura Pozzi,
Joseph Yeh, John Wawrzynek
Department of Electrical Engineering and Computer Sciences
University of California–Berkeley

SCORE is a programming model for reconfigurable computing designed for application longevity and scalability, based on a streaming dataflow compute model (Section 5.1.3) and employing several system architectures (Section 5.2) to support scalability. The compute model allows us to abstract away hardware details such as platform capacity (e.g., number of lookup tables [LUTs]) and the detailed cycle-by-cycle timing of hardware implementation. This allows a single application description to automatically run faster on larger hardware or to fit onto smaller hardware. The abstraction of platform size and clock cycle timing makes SCORE a higher-level programming model than RTL-level descriptions such as VHDL (Chapter 6). The streaming dataflow model allows high concurrency and natural task descriptions for a large class of streaming applications, including signal and image processing.

Figure 9.1 shows one of the key scaling forms enabled. We capture the computation as a streaming dataflow graph of persistent operators (Section 5.1.2) abstracted from a particular platform (Figure 9.1(a)). On small hardware platforms, we use a phased reconfiguration manager (Phased reconfiguration manager subsection of Section 5.2.2) to implement the task as a sequence of configurations on the available hardware (Figure 9.1(b)). For larger platforms, more operators can be placed spatially, exploiting greater concurrency to reduce runtime (Figure 9.1(c)).

To achieve scalability, Stream Computations Organized for Reconfigurable Execution (SCORE) allows and encourages the programmer to ignore the hardware capacity of a particular platform and focus on capturing the fully spatial, streaming dataflow graph. A combination of the compiler and the runtime system must decompose and schedule the application onto a variety of hardware capacities. To support late-bound, runtime adaptation to various hardware platforms, the SCORE runtime employs a paged reconfiguration discipline (Section 9.2.4).

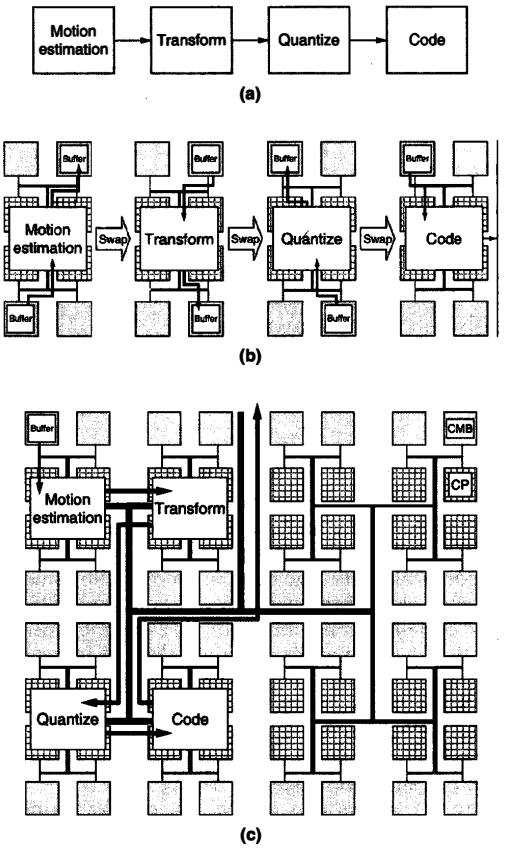


FIGURE 9.1 ■ Score application and sequential versus fully spatial execution: (a) a video compression task, (b) a capacity-limited sequential implementation, and (c) a fully spatial implementation on SCORE hardware.

In implementing this model, we must

- Provide concrete programming language instantiations for describing SCORE applications (Section 9.1).
- Select and employ suitable system architectures to implement the application and support area–time trade-offs for scalability (Section 9.2).
- Compile between the programming language description of the application and the runtime system architectures (Section 9.3).
- Provide runtime support for the tasks that must be performed during execution (Section 9.4).

The SCORE programming model demonstrates how compute model and system architectures come together to efficiently support a class of streaming applications.

9.1 PROGRAMMING

The specific compute model SCORE supports is Dynamic Streaming Dataflow with Allocation but without peeks (Dynamic streaming dataflow and Streaming dataflow with allocation subsections of Section 5.1.3), making it fully deterministic. Programs are composed by linking together operators (functions or objects, Section 5.1.2) and memory segments with first-in-first-out (FIFO) stream links (Section 5.1.3). Operators themselves can be described by their behavior or composed structurally as a graph.

Any number of languages that obey streaming dataflow semantics can be defined to program SCORE computations. The key requirements are to capture operators with appropriate dataflow input/output (I/O) interfaces and to allow operator compositions.

SCORE can be programmed with conventional programming languages (e.g., C++, Java) by defining stylized language subsets and library support to describe and compose SCORE operators. In Section 9.1.2, we show how to use C++ for dynamic composition.

In a multi-threaded language, such as Java or C++, with an appropriate thread package, a SCORE operator would be an independent thread that communicates with the rest of the program only through single-reader, single-writer I/O streams. Specifically, SCORE does not have a global, shared memory abstraction among operators (Single Memory Pool, Section 5.1.4). An operator may *own* a chunk of the address space (a memory segment) during operation and return it after it has completed, but no two operators may own a piece of memory simultaneously.

Alternately, SCORE programming could use a modern system-level design language, such as System C [1], as long as the communication library provides suitable dataflow communication semantics. To focus on the necessary semantics during SCORE development, we define an intermediate register transfer level (RTL) language to describe SCORE operators and their composition (Section 9.1.1). We view this intermediate language, TDF, as a device-independent, assembly language target on the way to platform-specific executable operators.

9.1.1 Task Description Format

Task Description Format (TDF) is basically an RTL-level operator description with special syntax for handling input and output datastreams from the operator [7, 22]. Common datapath operators can be described using a C-like syntax. For example, Figure 9.2 shows how an FIR computation might be implemented in TDF. Operators may have parameters whose values are bound at operator instantiation time; parameters are identified with the keyword `param`. In the

```
fir4(param signed[8] w0, param signed[8] w1,
     param signed[8] w2, param signed[8] w3,
     // param's bound at instantiation time
     input unsigned[8] x,
     output unsigned[20] y)
{
     state only(x): // "fire" when x present
     {
       // assignment to output y denotes a stream write
       y = w0*x + w1*x@1 + w2*x@2 + w3*x@3;
       // x@n notation picks out nth previous value for
       //   x on input stream.
       // (this notation is patterned after Silage [2])
       goto only; // loop in this state
     }
}
```

FIGURE 9.2 ■ A TDF specification of 4-TAP FIR (a static rate operator).

FIR example, the coefficient weights are parameters; these are specified when the operator is created, and the values persist as long as the operator is used. The FIR reads from a single input stream (`x`) and produces a single output stream (`y`); the assignment to `y` denotes the stream write. The behavior of the state is gated on the arrival of the next `x` input value, producing a new `y` output for each such input.

To allow dynamic-rate dataflow (Dynamic streaming dataflow subsection of Section 5.1.3), the basic form of a behavioral TDF operator is that of a finite-state machine (FSM) (Finite State, Section 5.1.4), in which each state specifies the inputs that must be present before it can fire. Once the inputs arrive, the operator consumes them, and the FSM may choose to change states based on the input data consumed. A simple `merge` operator is shown in Figure 9.3 to demonstrate how the state machine can also be used to allow data-dependent consumption of input values. (Note: This version has been simplified for illustration; it does not properly handle the end-of-stream condition.) Output value production can be conditioned as illustrated in the `uniq` example shown in Figure 9.4. Together, data-dependent input consumption and output production allow the user to specify arbitrary, deterministic, dynamic-rate operators.

Of course, the FSM gives the user the semantic power to describe heavily sequential and complex, control-oriented operators. Nonetheless, the programmer should avoid sequentialization and complex control when possible, as operators with many states are less likely to use spatial computing resources efficiently. Larger operators can be composed structurally from smaller operators in a straightforward manner, as shown in Figure 9.5.

9.1.2 C++ Integration and Composition

With a suitable stream implementation and interface code, SCORE operators can be instantiated by and used with a conventional, multi-threaded programming

```
signed[w] merge(param unsigned[6] w,
              // can use parameters to define data width
               input signed[w] a,
               input signed[w] b)
{

  signed[w] tmpA; // define local state inside the operator
  signed[w] tmpB;
  // states used here to show dynamic data consumption
  state start(a,b): // requires inputs on both a and b to be
                    //   available in order to evaluate
   {
     // assignments to local variables have C-like semantics
     tmpA=a;  tmpB=b;
     if (tmpA<tmpB) { merge=tmpA;
                      goto replaceA; }
     else { merge=tmpB;
            goto replaceB; }
     // note: assignment to function name signifies a write to the
     //  output stream which is returned from operator instantiation
   }
  state replaceA(a): // requires availability of only input a
   {
     tmpA=a;
     if (tmpA<tmpB) { merge=tmpA;
                      goto replaceA; }
     else { merge=tmpB;goto replaceB; }
   }
 state replaceB(b): // requires availability of only input b
  {
    tmpB=b;
    if (tmpA<tmpB) { merge=tmpA;
                     goto replaceA; }
    else { merge=tmpB; goto replaceB; }
  }
}
```

FIGURE 9.3 ■ A TDF specification of `merge` operator (a dynamic input rate operator).

language. Figure 9.6 shows an example C++ program that uses the `merge` and `uniq` operators defined in Figures 9.3 and 9.4. Note that SCORE operator instantiation and composition can be performed in C++ code. Once created, the operators behave as independently running threads, operating in parallel with the main C++ execution thread. In general, a SCORE operator will run until its input streams are closed or its output streams are released (i.e., the stream is deallocated with a `free`-like operation).

After primitive behavioral (or leaf) operators have been defined (e.g., in TDF or some other suitable form) and compiled into their hardware-level implementation, large programs can be composed entirely in a conventional programming language as just described and illustrated in Figure 9.6. If one thinks of TDF as a portable assembly language for critical computational building blocks, then this language binding allows a high-level language to compose these building blocks

```
// uniq behaves like the unix command of the same name;
//  it filters an input stream, removing any adjacent, duplicate
//  entries before passing them on to the output stream.
signed[w] uniq(param unsigned[6] w,
               input signed[w] x)
{
   signed[w] lastx;
   state start(x):
     { lastx=x; uniq=x; goto loop;}
   state loop(x):
     {
       if (x=!lastx)
         { lastx=x; uniq=x; }
       goto loop;
     }
}
```

FIGURE 9.4 ■ A TDF specification of `uniq` operator (a dynamic output rate operator).

```
merge3uniq(param unsigned[6] n,
           input signed[n] a,
           input signed[n] b,
           input signed[n] c,
           output signed[n] o)
{
         signed [n] t;
         t=merge(n,merge(n,a,b),c);
         o=uniq(n,t);
}
```

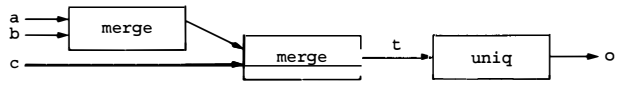
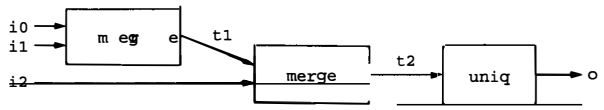


FIGURE 9.5 ■ The TDF compositional operator.

in much the same way that assembly language kernels are composed using high-level languages in order to efficiently program early DSPs and supercomputers. The instantiation parameters for TDF operators allow the definition of generic operators that can be highly customized to the needs of the application.

9.2 SYSTEM ARCHITECTURE AND EXECUTION PATTERNS

To support the SCORE programming model efficiently, implementations are based on several system architectures and execution design patterns (e.g., DeHon et al. [3]). In this section, we highlight how these architectures are used and introduce additional execution patterns.




```
#include "Score.h"
#include "merge.h"
#include "uniq.h"
int main()
{
  char data0[] = { 3, 5, 7, 7, 9 };
  char data1[] = { 2, 2, 6, 8, 10 };
  char data2[] = { 4, 7, 7, 10, 11 };
  // declare streams
  SIGNED_SCORE_STREAM i0,i1,i2,t1,t2,o;
  // create 8-bit wide input streams
  i0=NEW_SIGNED_SCORE_STREAM(8);
  i1=NEW_SIGNED_SCORE_STREAM(8);
  i2=NEW_SIGNED_SCORE_STREAM(8);
  // instantiate operators
  //  note: instantiation passes parameters and streams to the operators
  t1=merge(8,i0,i1);
  t2=merge(8,t1,i2);
  o=uniq(8,t2);
  // alternately, we could use: new merge3uniq (8,i0,i1,i2,o);
  // write data into streams
  //  (for demonstration purposes;
  //    real streams would be much longer and not come from main)
  for (int i = 0; i < 5; i++) {
    STREAM_WRITE(i0, data0[i]);
    STREAM_WRITE(i1, data1[i]);
    STREAM_WRITE(i2, data2[i]);
  }
  STREAM_CLOSE(i0); // close input streams
  STREAM_CLOSE(i1);
  STREAM_CLOSE(i2);
  // output results (for demonstration purposes only)
  for (int cnt=0; !STREAM_EOS(o); cnt++) {
    cout << "result["<< cnt << "]=" <<
       STREAM_READ(o) << endl;
  }
  STREAM_FREE(o);
  return(0);
}
```

FIGURE 9.6 ■ An example of instantiation and usage in C++.

9.2.1 Stream Support

SCORE heavily leverages the stream abstraction (Chapter 5, Section 5.1.3) for communication between operators. The streamed data can be assigned to a buffer if the producer and consumer are not coresident (see Figure 9.1(b));

if they are coresident, the data can be assigned to physical networking (see Figure 9.1(c)). Further, any number of mechanisms (e.g., shared bus, packet-switched network, time-multiplexed network, configured links) can implement the stream based on data rate, predictability, and platform capabilities. Once data communication is organized as a stream, the platform knows which data to prefetch and how to package it to or from memory.

When a SCORE implementation physically implements streams as wires between dynamic-rate operators, data presence (Data presence subsection of Section 5.2.1) tags allow us to abstract out dynamic data rates or delays. While data presence allows producers to signal consumers that data are not ready, it is often useful to signal the opposite direction as well; consequently, we also implement a *back-pressure* signal, which allows the consumer to inform the producer that it is not ready to consume additional inputs. We can further place queues between the producer and the consumer to decouple their cycle-by-cycle firing.

When the consumer is not ready, produced values accumulate in the queue, allowing the producer to continue operation; if there are stored values in the queue, the consumer can continue to operate while the producer is stalled as well. Queues are of finite size, so a full queue also uses back-pressure to stall an attached producer. In dynamic data rate operations where queue size cannot be bounded (Dynamic streaming dataflow subsection of Section 5.1.3), the hardware signals the OS when queues fill, and the OS may need to allocate additional queue capacity at runtime to prevent deadlock [4].

9.2.2 Phased Reconfiguration

When the operator graph is too large for the platform, it is necessary to share the physical hardware in time (see Figures 9.1(b) and 9.7). For a reconfigurable platform, this can be done by changing the configuration overtime, to implement the

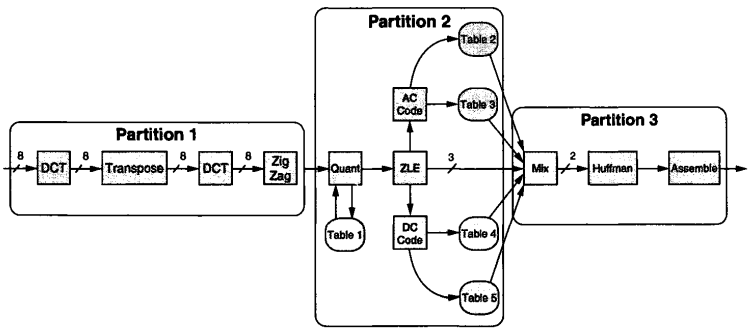


FIGURE 9.7 ■ Partitioning of a JPEG image encoder to match platform capacity.

graph in pieces (Phased reconfiguration manager subsection of Section 5.2.2). Reconfiguration, however, can be an expensive operation requiring many cycles. To minimize its overhead cost, we want to run each operator for many cycles between reconfigurations. In particular, if we can ensure that each operation runs for a large number of cycles compared to the reconfiguration time, then we can make the overhead for reconfiguration small ($T_{run\text{-}before\text{-}reconfig} >> T_{config}$). Streaming data with large queues helps us achieve this. We can queue up a large number of data items that will keep the operator busy. We then reconfigure the operator, compute on the queued data, and, if the consumer is not coresident, queue up the results (Figure 9.1(b)). When the input queue is empty or the output queue is full, we reconfigure to the next set of operators.

9.2.3 Sequential versus Parallel

When the platform contains both processors and reconfigurable logic, it is possible to assign some operators to the processor(s) (Processor subsection of Section 5.2.2) and some to the reconfigurable fabric. We can compile SCORE operators either to processor instructions or to reconfigurable configurations, and we can even save both implementations as part of the program executable. At load time or runtime, low-throughput operators can be assigned to the sequential processor(s), while high-throughput logic can be assigned to the reconfigurable fabric. As the size of the reconfigurable fabric grows, more operators can be implemented spatially on it.

Phased reconfiguration can be ineffective when mutually dependent cycles are large compared to the size of the platform. Processors are designed to time-multiplex their hardware at a fine granularity; thus, one way to fit large operator cycles onto the platform is to push lower throughput operators onto the processor until the cycle is contained.

We interface the processor to the reconfigurable array using a streaming coprocessor arrangement (Streaming Coprocessors, Section 5.2.1). The processor can write data into stream FIFOs to go to the reconfigurable array coprocessor, and it reads data back from them. This decouples the cycle-by-cycle operation of the reconfigurable array from the processor, abstracting the relative timing of the two units. In the case where the reconfigurable array can be occupied (e.g., allocated to another operator or task), this reduces coresidence requirements between operators on the array and processor. As a result, the options for the array size to vary among platform implementations increase.

9.2.4 Fixed-size and Standard I/O Page

To allow the platform size to vary with the implementation platform, it is necessary to perform placement at load time or runtime based on the amount of physical hardware and the time-multiplexed schedule. If we had to place everything at the LUT level, we would have a very large placement problem. Further, if we allowed partial reconfiguration in order to efficiently support the fact that different operators may need to be resident for different amounts of time, we

would have a fragmentation and bin-packing problem [5], as different operators take up different space and have different footprints. We can simplify the runtime problem by using a discipline of fixed-size pages that have a standard I/O interface.

First, we decide on a particular page size (e.g., 512 4-LUTs) for the architecture. At compile time, we organize operators into standard page-size blocks so that we can perform the intrapage placement and routing offline at compile time. At runtime, we simply place pages and perform interpage routing. The runtime placement problem is simplified because all pages are identically sized and interchangeable. Furthermore, because pages are typically 100 to 1000 4-LUTs, the runtime placement problem is two to three orders of magnitude smaller than LUT-level placement. Unfortunately, fixed-size pages may incur internal fragmentation, leaving some resources in each page unused. Brebner's SLU is an early example of this pattern [6].

Note that this is the same basic approach used in virtual memory, where we do not manage every bit or even every word independently, but instead gather a fixed number of words into a page and manage (e.g., map and swap) them as a group. In both cases, this reduces the overhead associated with page mapping considerably.

9.3 COMPILATION

We have developed a complete compilation flow from TDF to conventional FPGAs using Verilog (an HDL similar to VHDL—see Chapter 6) as an intermediate form (Figure 9.8) [7]. The TDF compiler, `tdfc`, automatically generates RTL Verilog to efficiently implement the streaming constructs of the TDF language, including flow control checking, stream buffering in queues, and stream pipelining. The TDF compiler also maps between abstract operators of arbitrary size and the fixed-size pages supported at runtime by the system architecture.

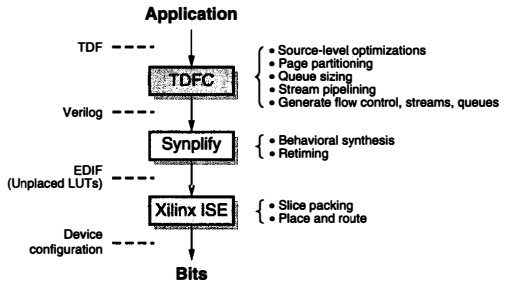


FIGURE 9.8 ■ TDF compilation flow targeting an FPGA.

The compiler then emits a netlist of pages for compilation by a commercial backend FPGA synthesis, place, and route flow.

Because SCORE streams abstract the number of clock cycles between operators, we can pipeline both the interconnect between operators and the operator datapaths. To pipeline operators, the compiler adds registers to the input and output streams and employs retiming (Chapter 18) to redistribute the registers into the operator logic.

To accommodate the wide range of operator sizes that the programmer may produce, the compiler must perform operator packing and splitting in order to target any particular, fixed-size page. Our previous experience suggests that most user-written leaf operators require fewer than 512 4-LUTs, which means that page packing will be adequate to reshape most applications. Many large operators are feedforward pipelines (e.g., DCT, IDCT), which can be easily decomposed using directional cuts in the dataflow. For the general case, it is necessary to decompose large state machines to fit them onto small pages. This could be done by starting with individual states and clustering state logic and datapaths, obeying the page area and I/O bound. To minimize delay, the goal is to group states that typically execute together so as to minimize the frequency of state transitions that cross the page boundary. Clustering techniques such as those described by Li et al. [8] can be employed for this general clustering case.

9.4 RUNTIME

To support the late-bound task and platform mapping integral to SCORE's power and scalability, we must perform scheduling, placement, and routing no earlier than load time. In this section, we highlight how these tasks can all be performed quickly at load time or runtime.

9.4.1 Scheduling

We support SCORE's virtualization model in the presence of late-bound platform mapping with a load-time and runtime scheduler. We do not know the capacity of the platform until load time; consequently, we cannot partition the graph into sets of pages that fit on the platform before then. Further, because operators have dynamic execution times and dynamic consumption and production rates, the relative execution time of each operator cannot be known with certainty until execution. To support SCORE efficiently, we must be able to:

- Quickly partition the page graph into platform-feasible components (within milliseconds).
- Produce a high-quality schedule—that is, one that minimizes the time to run the task (minimizes the make span).
- Minimize the sequential handling required for managing reconfiguration and advancing the schedule.

In the simplest cases, we partition the graph once, at load time, when the program starts and never again. In this way, we amortize the cost of partitioning across the entire application runtime (Figure 9.9). If the application will run for seconds, we can afford tens of milliseconds for this scheduling operation while keeping the overhead small. If we can decrease the scheduling time, then it will be possible to run even shorter jobs efficiently. In more advanced cases, the graph may change during execution, or the execution rates of operators may change in a data-dependent way. In such cases, it might be useful to repartition and reschedule the graph during execution. The shorter we can make the partitioning time, the more frequently we can afford to invoke the partitioner without paying a large overhead.

We have developed a series of schedulers to address these issues [9–12]. Our highest-quality scheduler (shown in Figure 9.10) is quasi-static and load-time based [10], and operates in two phases: (1) load-time partitioning and (2) runtime schedule advancement. At application load-time, the scheduler partitions the page-level dataflow graph into platform-feasible subgraphs. This partition can use feedback information on operator and stream activity rates based on previous program runs. The load-time partitioning heuristic requires only a few hundred thousand processor cycles (e.g., submillisecond time on gigahertz processors) for graphs with up to one hundred operators [12].

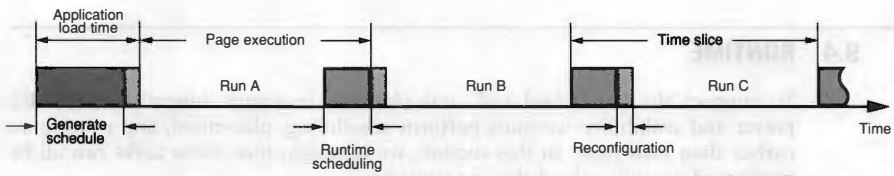


FIGURE 9.9 ■ An application execution timeline.

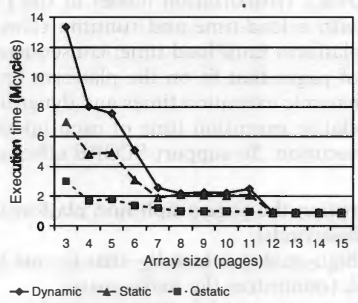


FIGURE 9.10 ■ JPEG decoder scaling: Total execution time is compared among fully dynamic, fully static, and quasi-static schedulers.

The result is a schedule for the phased reconfiguration. During execution, the runtime system advances the computed schedule by reconfiguring the array at regular intervals (Phased reconfiguration manager subsection of Section 5.2.2). The schedule computed at load time specifies a nominal period for each schedule timeslice. Additionally, the system monitors execution to determine when the current configuration can no longer make forward progress (e.g., all input buffers are empty or all output buffers are full) and dynamically triggers early phase termination and schedule advancement.

9.4.2 Placement

Using the fixed-size and standard I/O pages discipline (Section 9.2.4) we immediately reduce the size of the placement task by two to three orders of magnitude. Nonetheless, the placement task may still take too long when run using conventional single-processor-based placers at reconfiguration time or even load time. Fortunately, once we have a spatially parallel reconfigurable computing platform, we can use the platform itself to perform placement substantially faster. In Wrighton and DeHon [13] and Wrighton [14], we show how to perform simulated annealing spatially with reconfigurable logic; we can place a graph with 1000 movable elements in roughly 1 million cycles. Even if we only ran the placement engine at 100 MHz, this would mean that we could perform placement in 10 ms. If each page held 512 4-LUTs, this would correspond to platforms with half a million 4-LUTs.

The key idea for spatial simulated annealing is to build a placement engine on top of the reconfigurable platform. If we make each page large enough, then it can act as a cellular placement cell. As a placement cell, it holds a candidate, logical page and negotiates exchanges with its nearest neighbors (i.e., cellular automata system architecture Section 5.2.5). A pair of adjacent pages will swap logical pages if they estimate that the swap will produce a superior placement (e.g., shorter wire lengths) or if the randomness in the simulated annealing process suggests attempting the swap anyway. All pages can be paired up and can negotiate swaps in parallel, allowing many moves per swap epoch.

By pairing up only neighbors, we can guarantee minimizing the interconnect for this placement engine and keep the cycle times short. Because there is one cellular placement cell for every page site on the device, the hardware and parallelism in the placement engine scales exactly to the size of the placement problem that needs to be solved. Wrighton and DeHon [13] estimate that 400 4-LUTs are adequate to implement a 100 MHz cellular placement cell on Xilinx Virtex-II–generation hardware [15]; this suggests that SCORE platforms with 512 LUT pages will be able to perform their own placement.

9.4.3 Routing

Once the pages have been placed, we must perform interpage routing. Again, we can exploit the fact that we have a spatially parallel computing platform to route tasks in 100,000 to 1 million cycles [16]. Here, we augment the interpage network with additional logic to allow it to identify all free paths between

a source node and a sink node in parallel. This permits a flooding search (e.g., Figure 9.11) to find a free path in the time it takes to propagate a signal across the network rather than the time it takes to perform a sequential search on a large graph structure in memory. Consequently, each new path can be added in tens of cycles rather than the tens of thousands of cycles required by the best software routers.

Using randomization, rip-up, and multiple restarts, this approach can even perform congestion negotiation and achieve comparable quality to PathFinder [17] (Chapter 17), the state-of-the-art software-routing algorithm for FPGAs [18, 19]. With word-wide (e.g., 16-bit) datapaths for the interpage network, the additional area overhead for this augmented network is less than 30 percent when network routing channels are switch-area limited; the augmented network adds only control wires, so it has almost no area overhead when network-routing channels are wire dominated.

An alternate approach is to employ a packet-switched network for interpage routing (see Marescaux et al. [20] and Kapre et al. [21]) to avoid the need to compute and configure the network. Packet switches are generally much larger and have higher latency than configured switches, but they may be able to handle multirate and dynamic traffic more efficiently.

Figure 9.11 shows the result of a path search for a route from node 4 to node 2. Light thick lines show preexisting routes; dark thick lines show the free paths explored between source and sink. At the crossover switchbox (labeled "XXX"), only a single switch is found by both source- and sink-initiated searches.

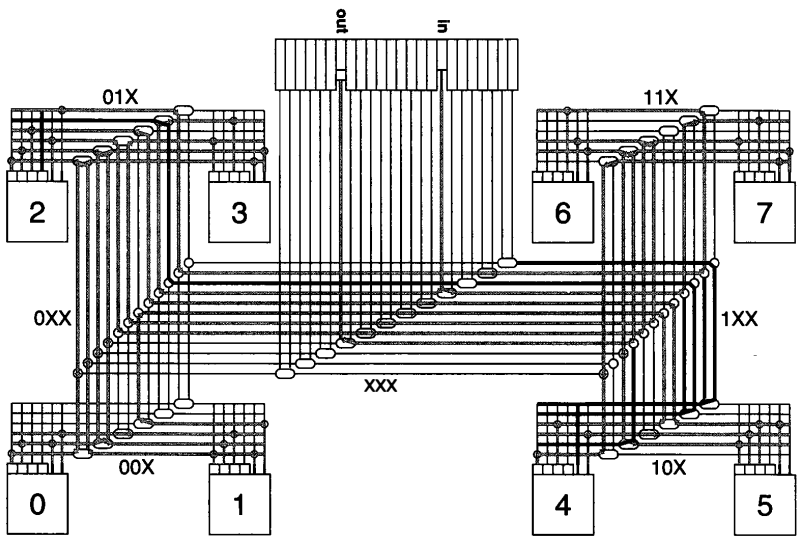


FIGURE 9.11 ■ A spatially parallel path search.

9.5 HIGHLIGHTS

SCORE compilation has automatically mapped image-processing applications (e.g., wavelet, JPEG, MPEG) to streamed implementations that exceed 100 MHz sample throughput on a Virtex-II Pro XC2VP70-7 [12]. In comparable technology, a 4-page SCORE design outperforms a Pentium-3 (500 MHz) by 10 times on JPEG compression. Mapped design performance scales to deliver larger speedup with additional pages (see Figure 9.10).

For further details on SCORE, see DeHon et al. [12] and Caspi et al. [22].

References

[1] Open System C Initiative. *System C 2.1 Language Reference Manual*, May 2005 (*http://www.systemc.org*).

[2] D. Genin, J. Rabaey, P. Hilfinger, C. Scheers, H. DeMan. DSP specification using the SILAGE language. *Proceedings of the IEEE ICASSP Conference*, April 1990.

[3] A. DeHon, J. Adams, M. deLorimier, N. Kapre, Y. Matsuda, H. Naeimi, M. Vanier, M. Wrighton. Design patterns for reconfigurable computing. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[4] T. M. Parks. *Bounded Scheduling of Process Networks*, UCB/ERL95–105, University of California, Berkeley, 1995.

[5] K. Bazargan, R. Kastner, M. Sarrafzadeh. Fast template placement for reconfigurable computing systems. *IEEE Design and Test of Computers* 17(1), January–March 2000.

[6] G. Brebner. The swapable logic unit: A paradigm for virtual hardware. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 1997.

[7] E. Caspi. *Design Automation for Streaming Systems*, Ph.D. thesis, University of California, Berkeley, 2005.

[8] Z. Li, K. Compton, S. Hauck. Configuration caching techniques for FPGAs. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

[9] M. Chu. *Dynamic Runtime Scheduler Support for SCORE*, Master's thesis, University of California, Berkeley, December 2000.

[10] Y. Markovskiy, E. Caspi, R. Huang, J. Yeh, M. Chu, J. Wawrzynek, A. DeHon. Analysis of quasi-static scheduling techniques in a virtualized reconfigurable machine. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, February 2002.

[11] Y. Markovskiy. *Quasi-Static Scheduling for SCORE*, Master's thesis, University of California, Berkeley, December 2004.

[12] A. DeHon, Y. Markovskiy, E. Caspi, M. Chu, R. Huang, S. Perissakis, L. Pozzi, J. Yeh, J. Wawrzynek. Stream computations organized for reconfigurable execution. *Journal of Microprocessors and Microsystems* 30(6), September 2006.

[13] M. Wrighton, A. DeHon. Hardware-assisted simulated annealing with application for fast FPGA placement. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, February 2003.

[14] M. Wrighton. *A Spatial Approach to FPGA Cell Placement by Simulated Annealing*, Master's thesis, California Institute of Technology, June 2003.

[15] Xilinx, Inc. *Xilinx Virtex-II 1.5V Platform FPGAs Data Sheet*, San Jose, July 2002.

[16] A. DeHon, R. Huang, J. Wawrzynek. Hardware-assisted fast routing. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2002.

[17] L. McMurchie, C. Ebeling. PathFinder: A negotiation-based performance-driven router for FPGAs. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, February 1995.

[18] R. Huang, J. Wawrzynek, A. DeHon. Stochastic, spatial routing for hypergraphs, trees, and meshes. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, February 2003.

[19] A. DeHon, R. Huang, J. Wawrzynek. Stochastic spatial routing for reconfigurable networks. *Journal of Microprocessors and Microsystems* 30(6), September 2006.

[20] T. Marescaux, V. Nollet, J.-Y. Mignolet, A. Bartic, W. Moffat, P. Avasare, P. Coene, D. Verkest, S. Vernalde, R. Lauwereins. Run-time support for heterogeneous multitasking on reconfigurable SOCs. *INTEGRATION, The VLSI Journal* 38(1), October 2004.

[21] N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, A. DeHon. Packet-switched vs. time-multiplexed FPGA overlay networks. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2006.

[22] E. Caspi, M. Chu, R. Huang, N. Weaver, J. Yeh, J. Wawrzynek, A. DeHon. Stream Computations Organized for Reconfigurable Execution (SCORE): Introduction and tutorial (*http://www.cs.berkeley.edu/projects/brass/documents/score_tutorial.html*); a short version appears in *FPL '2000* (Lecture Notes in Computer Science, 1896), 2000.

CHAPTER 10

PROGRAMMING DATA PARALLEL FPGA APPLICATIONS USING THE SIMD/VECTOR MODEL

Maya B. Gokhale
Lawrence Livermore National Laboratory

In the Single Instruction Multiple Data (SIMD) model, aggregate operations on arrays and vectors can be mapped to arrays of function units. A single instruction stream is dispatched from a control unit to the function units, which operate in lockstep on the data sequences. Reconfigurable hardware is well suited to perform SIMD (also called *vector* or *data parallel*) computation (see Section 5.1.5). Groups of lookup tables (LUTs) can be configured as function units, and the data local to each unit can be stored in distributed memories. This chapter explores parallel processing on reconfigurable computers using the SIMD/vector model.

Reconfigurable computers can exploit parallelism at many different levels of granularity, from coarse-grained parallel tasks to fine-grained instruction-level parallelism. The massive amount of parallelism available in the reconfigurable computer more than compensates for its slow clock rate—one-tenth the clock rate of modern microprocessors. Raw spatial parallelism is plentiful in reconfigurable processors, especially those based on FPGAs. The challenge is to partition and map the application onto the inherently parallel fabric of lookup tables, DSP blocks, and memories. Parallel activities can be explicitly described and scheduled by the programmer or hardware designer, or can be inferred through analysis of the source code. SIMD/vector parallelism is very well suited to the spatial parallelism of FPGAs and other coarse-grained arithmetic logic units (ALU) arrays. In this programming model, aggregate data such as vectors and matrices are processed in parallel on arrays of function units.

10.1 SIMD COMPUTING ON FPGAS: AN EXAMPLE

As an introduction to SIMD computing on FPGAs, Figure 10.1 shows an SIMD array customized to perform two vector operations. A vector A is scaled by a constant factor, and then the dot product $A \cdot B = \Sigma a_i \times b_i$ of vectors A and B is performed. In this example, the number of SIMD processors is equal to the size of the vectors. Each processor holds one element of A and one of B. There is an additional storage location in each processor to hold the result of the $\times$ operation.

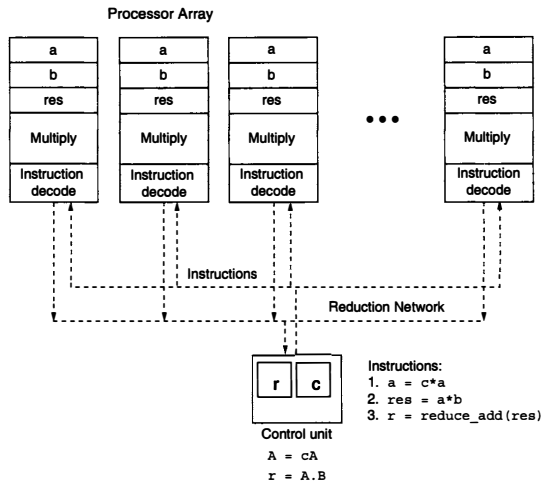


FIGURE 10.1 ■ An SIMD dot-product machine.

The control unit generates the instruction stream. An instruction can be executed on the control unit itself, on each processor of the processor array, or cooperatively on both. In this example, the control unit sends the constant *c* as part of the first multiply instruction. The constant appears as an immediate operand in the instruction to each processor. Next the control unit sends the second multiply instruction to the processor array, and all processors perform the operation `res=a*b`. The final instruction performs a *reduction*, a global combining operation, in which each processor sends its instance of `res` into the reduction network. Because the operation is a global sum, all the `res` instances are summed and the result is stored in the control unit variable *r*. While the example shows the control unit sending three separate instructions to the processor array, on an FPGA it is very possible that the controller will send a single instruction that results in a multi-cycle sequence of multiply operations followed by the global sum.

In this idealized example, the number of processors exactly matches the size of the vectors. In real applications, there are many different vectors of different sizes. The vectors must be distributed to the processors in blocks, and each processor must multiply subvectors of elements. If the number of processors doesn't evenly divide the vector size, some processors must remain inactive when the tail ends of the vectors are multiplied. Each processor must keep a subaccumulation, and, when the entire vector has been processed, the global sum is performed over the partial sums. When the processor array is on an FPGA, the compiler must synthesize state machines (FSMD subsection of Section 5.2.2) to

control the sequence of operations and iterate over the blocks of data. Designing algorithms for reconfigurable computers in the SIMD model in the face of these real-world complicating factors will be addressed in Section 10.4.

10.2 SIMD PROCESSING ARCHITECTURES

SIMD/vector machines were among the first parallel processors to be designed. From the days of the Iliac IV, with 64 processing elements (PEs) receiving instructions from a control processor, this parallel-processing architecture has gone through myriad incarnations. Notable among SIMD arrays are the Connection Machine, which had thousands of simple PEs operating in synchrony [1], as well as DAP and MasPar (late 1980s [2]). The Terasys Integrated Circuit [3] and the Clearspeed SIMD array [4] both included an SIMD processing array on a single integrated circuit.

Historically, supercomputers with dedicated floating-point function units used for processing arrays and vectors were called vector supercomputers, while massively parallel, highly interconnected arrays of function units were referred to as SIMD, or data parallel. More recently, as small arrays of function units have been incorporated into the architecture of scalar processors, the terms *SIMD*, *vector*, and *data parallel* have become interchangeable. This is especially apropos to reconfigurable computers, in which arbitrary numbers and types of function units may be used with many different kinds of interconnect patterns.

An SIMD processing array, illustrated in Figure 10.2, consists of a collection of identical processing elements operating in lockstep. The PEs all execute exactly the same instruction, which is broadcast to them from a control unit, or "sequencer," as indicated by the dotted lines in the figure. Each PE has a local memory from which to fetch data operands and store results. On an SIMD array, control flow instructions, such as branching, conditional branch, and subroutine call, are executed on the control unit.

Data-dependent branching represents a particular challenge when different instances of the data are resident in each PE's memory. Depending on the data value, some PEs might evaluate the branch predicate to true and others to false. Because they all must execute the same instruction at the same time, each PE has a predicate mask flag (the `M` in the corner of each ALU) indicating whether the PE should execute or ignore the current instruction.

The PE sets the predicate mask to the result of evaluating the predicate on its data items, and then either executes subsequent instructions or is inactive. The control unit can reset PEs to the active mode by issuing "unconditional" instructions to them, directing them to ignore the predicate mask. The notion of predicated instructions, which is essential to SIMD processing, is also used in some microprocessor instruction sets [5], particularly in wide-word explicitly parallel architectures.

In SIMD processing, PEs exchange data synchronously. The PE interconnection network may be arranged as a linear array, as in Figure 10.2, or as a two-dimensional (or even three-dimensional) mesh or torus. In addition to

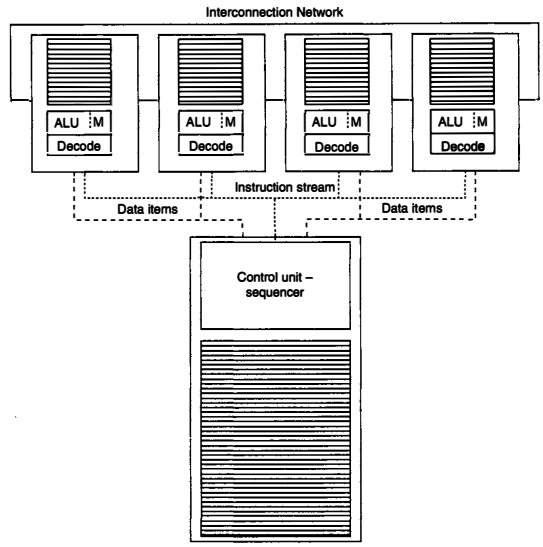


FIGURE 10.2 ■ An SIMD processing array.

nearest-neighbor communication (illustrated with solid lines in the figure), data parallel arrays usually include global combining networks for global reduction (sum, product, min, max, and logical) operations. The control unit can retrieve data from the memory of individual PEs and can also receive the result of the global combining operations (dashed lines in the figure).

A global combining network is illustrated in Figure 10.3, which shows a network organized as a binary tree with a combining operator at each interior tree node. Global combining networks can be used for any associative operation. With parallel tree operations, an `O(n)` operation is reduced to `O(log(n))`.

10.3 DATA PARALLEL LANGUAGES

High-level data parallel languages for SIMD machines were popularized in the late 1980s with the emergence of the Connection Machines CM-1, CM-2, and CM-5, and were adopted by other vendors. In the CM approach, a base language such as Fortran or C was extended with new keywords, syntax, and semantics. In the C* language, a data parallel extension to ANSI C, new data type modifiers `mono` and `poly` were introduced. A `mono` variable resides in the control unit memory, while a `poly` variable occupies memory local to each PE,

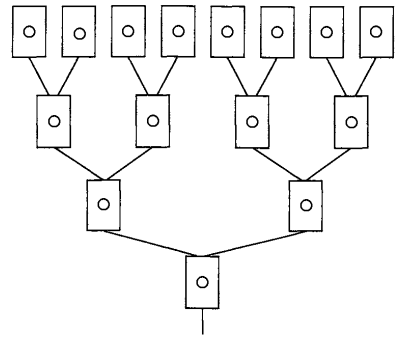


FIGURE 10.3 ■ A global sum network.

implicitly defining a vector or higher-dimension array. Operation on a `mono` variable is performed on the control unit, while a `poly` expression is evaluated independently on each PE.

Also in the 1980s, new syntax and intrinsic functions were introduced to express global combining operations, inter-PE communication, and unconditional execution.

Declaration of `poly` variables in most data parallel languages implicitly defines an aggregate object whose length is the number of PEs in the physical array. Unfortunately, most datasets do not conform in size or shape to the physical PE array, and therefore the programmer must arrange the data arrays in blocks distributed among the PEs' memories, and then loop over the blocks on each PE. The Connection Machines, however, supported "virtual" processors in microcode. The programmer could define an array of processing elements larger than the size of the physical PE array that better matched the size of the datasets, and microcode in each PE looped over the block of data in its memory.

10.4 RECONFIGURABLE COMPUTERS FOR SIMD/VECTOR PROCESSING

In contrast to specific physical implementations of SIMD arrays in silicon, a large variety of data parallel machines may be mapped onto FPGA-based reconfigurable computers. The data parallel model maps naturally to the physical structure of FPGAs, with dedicated hardware blocks of arithmetic units and memories tiled regularly in a two-dimensional array, as well as a flexible interconnect. In addition, there are many degrees of freedom in an FPGA implementation. The data parallel engine can be customized to the datasets being processed in terms of geometry (one versus multidimensional arrays), interconnect (linear, mesh, torus), and even PE instruction set.

An early experiment in data parallel computing on FPGAs was the dbC project [6] in which a data parallel language was compiled onto the Splash 2 reconfigurable logic array [7]. dbC was modeled on the Connection Machines' C* language. Like C*, dbC included the `mono` and `poly` data type modifiers to denote data on the control unit and SIMD array, respectively.

The size of the SIMD array could be specified at the language level by setting a predefined variable to the number of PEs. The linear array thus defined was automatically partitioned among the 16 FPGAs of the Splash system.

Instructions were broadcast to the FPGAs from the Sun workstation host, which served as the control unit. Unlike conventional SIMD arrays, the PE instruction set was not fixed. Rather, the compiler created a unique instruction set for each dbC program, generating a behavioral VHDL module (see Chapter 6) that was synthesized through the normal CAD tool flow. An instruction, rather than being a simple arithmetic or load/store operation, was synthesized as a predicated block. This could be a simple basic block—a straight-line sequence of code with a single entry and a single exit. If the C code contained `if` statements, the compiler transformed control dependence into data dependence [8], creating sequential predicated blocks that contained first the true branch and then the false branch of the `if`. Thus, a single instruction dispatched from the control unit to the SIMD array could result in a multi-clock-cycle block of logic executing a predicated hyperblock.

To exploit the flexibility of FPGAs to perform arithmetic on arbitrary bit-length operands, dbC allowed `poly` variables to be of user-specified bit length. dbC extended C integer data types by permitting C bit field syntax to be used to define the bit length of signed and unsigned integer variables. This ability was particularly valuable on early FPGAs with limited logic and interconnect. The arithmetic units synthesized within the SIMD PE were customized to the precision required, and the programmer specified that precision by the choice of data types.

In keeping with the SIMD interprocessor communication model, a runtime hardware library was built to implement global communications instructions such as min/max and a small set of logic operations, which were performed bit-serially by the Splash 2 control FPGA.

The dbC language and compiler thus combined a parallel language, traditional compiler transformations, and a simple form of hardware synthesis to generate a control program and FPGA bitstream for the Splash system.

To illustrate the dbC data parallel language and its mapping onto FPGAs, Figure 10.4 expands on the vector multiply example in Section 10.2. Line 3 illustrates the use of bit field syntax to define a new data type, a 24-bit integer, `my_int`. `DBC_net_shape` (line 6) is a predefined variable used to set the number of processors and their shape. (On Splash, the shape was limited to a linear array.) The vector multiply is divided into two sections. First there is a loop over the blocks of vectors resident on each PE (lines 31–34). The control unit handles the loop control and iteratively issues instructions in the loop body to the SIMD array. The += operation on line 33 is executed by each PE and accumulates the partial product into the `poly` variable `res`.

```
1   #define ISIZE 24
2
3   typedef poly int my_int:ISIZE;
4
5   /* specify 64 processors in a linear array */
6   unsigned in DBC_net_shape[1] = {64};
7
8   /* Each PE can hold up to 500 elements of the vector,
9      so maximum vector size is 500*64                    */
10
11  #define VEC_MAX 500
12  void main() {
13
14    /* vectors A, B, res are on each PE */
15    poly my_int A[VEC_MAX];
16    poly my_int B[VEC_MAX];
17    poly my_int res[VEC_MAX];
18
19    /* r, c, and vec_size are on the control unit */
20    mono unsigned long long int r;
21    mono int c;
22    mono int vec_size;
23    int i;
24
25    /* first initialize vec_size, vectors A and B, constant c */
26
27    /* next, compute vector multiply on the vector elements up to
28       the index that evenly divide the total number of PEs.    */
29
30    res = 0;
31    for (i=0; i<vec_size/DBC_nproc; i++) {
32      A[i] = A[i] * c;
33      res += A[i] * B[i];
34    }
35
36    /* now multiply the remaining elements of the vectors */
37
38    if (DBC_iproc < vec_size % DBC_nproc) {
39      A[i] = A[i] * c;
40      res += A[i]*B[i];
41    }
42
43    r += res;
44
45    /* continue computation */
46
47  }
```

FIGURE 10.4 ■ A vector multiply program in dbC.

The second section of code finishes the multiplication of final residue, potentially on a smaller number of PEs (lines 38–41). The `if` statement on line 38 sets the predicate mask bit to true in each PE whose processor number is less than the number of remaining elements of the vectors, and to false in all the other

PEs. The comparison of `vec_size` to `DBC_nproc` involves only `mono` variables and so is performed on the control unit and sent to the PE array as a constant in the instruction. Line 43 is a global accumulation of intermediate results from each PE into the control unit variable `r`.

There are some unique aspects to compiling SIMD algorithms to FPGA-based reconfigurable computers. For one, the compiler can synthesize an instruction set customized to the application. In our example, there need be only three instructions:

- `A[i] = A[i] * c; res += A[i] * b[i];`
- mask bit ← `DBC iproc < vec size % DBC nproc`
- `r += res;`

For another, the ALU can be customized to the operations used in the code. In this example, only a 24-bit multiplier, adder, and comparator are required. If different precision is needed, the PE can be resynthesized. In fact, if floating-point data types are necessary, floating-point, rather than integer arithmetic units can be instantiated. Finally, the PE array can be easily resynthesized to hold more or fewer PEs.

10.5 VARIATIONS OF SIMD/VECTOR COMPUTING

The SIMD programming model is attractive in its simplicity of parallel operation. There is a single instruction stream; inter-PE communication is global and synchronous; and the global reduction operations allow operations across the entire PE array. However, SIMD also has some deficiencies. Often there are cases in which some PEs perform slightly different operations than others, particularly with boundary conditions. The SIMD model requires that all PEs participate in all alternatives. This can result in poor performance in the presence of deeply nested `if` statements, as the instruction stream follows all possible control flows. For this reason, SIMD processing is often used in conjunction with other programming models on reconfigurable computers.

10.5.1 Multiple SIMD Engines

It is possible to map multiple SIMD engines onto an FPGA, with a controller for each engine synthesized in the reconfigurable logic. Such a system is illustrated in Figure 10.5. This capability was offered by the Fabric-based System [9], and demonstrated on a system-on-a-chip using the Altera Excalibur FPGA. In this framework, the on-chip microprocessor controls a flexible, runtime reconfigurable computing fabric of mesh-connected processing cells. Each cell has a separate local data memory and a small program memory that holds DSP-like microcode instructions. In SIMD mode, a group of cells all contain the same program and are sequenced through it by a customized control unit that is also in the reconfigurable logic.

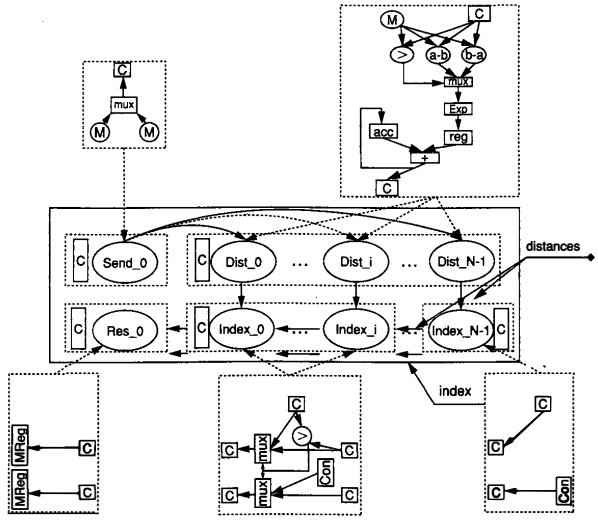


FIGURE 10.5 ■ An extended SIMD architecture.

The fabric illustrated in Figure 10.5 shows a multi-SIMD implementation of a compute-intensive kernel of the K-Means clustering algorithm. In this iterative data-mining algorithm, the dataset is partitioned into a predetermined number of classes. Initially, elements of the dataset are randomly assigned to classes, and a center C_i (where i ranges over the number of classes) of each class is computed. Then, for each element E_j (j ranges over the number of data elements) and each class C_i, the distance between E_j and C_i is computed. E_j is moved to the class closest to it in the distance metric, and the process repeats either for a fixed number of iterations or until there is no change from the previous iteration. In the example, the distance metric is $|E_j - C_i|$ (i.e., the absolute value [10] is used).

This design (Figure 10.5) implements the distance calculation, in which the distance between E_j and C_i is computed, and finds the class closest to each element E_j. There are five cell types—send, distance calculation, two for index calculation, and receive—each with its own control unit (labeled "C" in the figure). The "Dist" SIMD engine controls the distance calculation PEs; the "Index" SIMD engine, the index calculation PEs. The last index calculation has its own controller because its interconnect is slightly different from the others. Similarly, the send and receive cells have unique datapaths, so each has a dedicated controller.

In the figure, the computation is parallelized across classes, with one distance/index pair per class. A microprocessor controls the outer K-Means loop and updates class centers by loading new values into the `Send_0` cell's local memory. `Send_0` reads from one of two memories and sends the data element

out its communication channel. This allows the microprocessor to load one memory while the fabric is computing with the other. The distance calculation cells compute the distance between the pixel and the class centers. Their datapath is shown in the upper right box. The index calculation cells calculate the index of the class having the minimum distance to the pixel (the middle and right boxes at the bottom). The receive cell (`Res_0`) stores the class index corresponding to the minimum distance. It accepts data from two channels and writes into two memories.

Thus, an efficient parallel architecture for the K-Means clustering algorithm combines two SIMD arrays with three additional specialized processing units and a control microprocessor.

10.5.2 A Multi-SIMD Coarse-grained Array

In addition to FPGA-based data parallel systems, the Morphosys system [11] was designed as a coarse-grained SIMD array. Morphosys was an 8×8 array of reconfigurable logic cells controlled by a small RISC processor. Each row or column of the array operated in SIMD mode, executing the same instruction on different data instances. The RISC processor could dynamically load configurations into the array on a row/column granularity. This versatility in data parallelism and dynamic reconfiguration made it possible to map a combination of data parallel and control parallel algorithms onto Morphosys.

10.5.3 SPMD Model

A popular generalization of SIMD is the Single Program Multiple Data (SPMD) model (see Single program multiple data subsection of Section 5.2.4 and [12]) in which all processes independently execute the same program and can take different paths through it. SPMD differs from SIMD in that, rather than execute a global, synchronized communication step, programs use send/receive message passing to communicate with each other, and may employ other synchronization primitives such as barrier synchronization, in which each process waits at the barrier until all processes have reached it in their control flow.

SPMD is most common in parallel processing clusters. However, elements of it have also been adapted to FPGA computing. For example, in the Streams-C language, a CSP-like [13] parallel programming language for FPGAs [14], the programmer can define a parallel processor composed of an "array of processes," with each having the same hardware logic and control program, operating independently from the others, and using unidirectional channels to communicate.

10.6 PIPELINED SIMD/VECTOR PROCESSING

Pipeline processing can often be incorporated into SIMD/vector reconfigurable computing. This technique in essence synthesizes customized vector units that are replicated on the FPGA. Pipelined SIMD processing is especially beneficial on

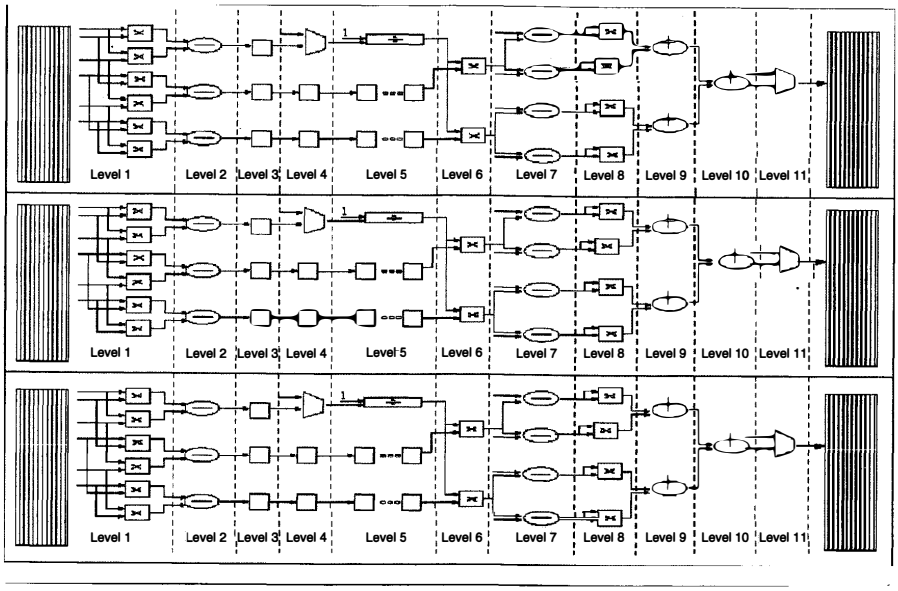


FIGURE 10.6 ■ SIMD with pipelined vector units.

FPGAs when complex arithmetic operations such as floating-point calculations must be performed.

Figure 10.6 shows an SIMD pipelined processing system in reconfigurable logic [15] having three PEs. Three pipelines are instantiated, with each receiving input parameters from a local memory and returning results to another memory. Each pipeline has 11 stages of floating-point operations, and each floating-point operation is, in turn, pipelined, resulting in a 43-stage pipeline. This pipeline implements the inner loop of a Monte Carlo simulation of radiative heat transfer in a two-dimensional chamber. In this case, three single-precision floating-point pipelines could be accommodated on a Xilinx Virtex-II Pro 100.

10.7 SUMMARY

In the SIMD/vector model, a tightly coupled ensemble of processors execute a single instruction stream issued by a control unit. The model can be synthesized onto an FPGA fabric. Having programmable hardware makes it possible to synthesize an instruction set tailored to the specific computations in the application. Customized data widths are naturally accommodated, as there is no fixed-width

ALU. Global combining operations utilizing parallel prefix networks can also be synthesized.

On FPGAs, the SIMD/vector model can be flexibly extended. Collections of SIMD subunits can be assembled and interconnected. This permits portions of the application that map naturally to the SIMD programming model to use it while still allowing other more irregular, control flow-dominated code to be synthesized on the same device. Pipeline processing can also be incorporated into the SIMD/vector processor, increasing the spatial parallelism available to the application.

Acknowledgments The contributions of Christophe Wolinski to Section 10.5 and of Jan Frigo to Section 10.6 are gratefully acknowledged.

References

[1] W. D. Hillis. *The Connection Machine*, MIT Press, 1989.
[2] R. M. Hord. *Parallel Supercomputing in SIMD Architectures*, CRC Press, 1990.
[3] M. Gokhale, B. Holmes, K. Iobst. Processing in memory: The Terasys massively parallel PIM array. *IEEE Computer* 23–31, 1995.
[4] Clearspeed. *http://www.clearspeed.com/*.
[5] D. I. August, et al. Integrated predicated and speculative execution in the IMPACT EPIC architecture. *International Symposium on Computer Architecture*, 1998.
[6] M. Gokhale, B. Schott. Data parallel C on a reconfigurable logic array. *Journal of Supercomputing* 9(3), 1995.
[7] D. A. Buell, J. M. Arnold, W. J. Kleinfelder (eds.). *Splash 2: FPGAs in a Custom Computing Machine*, Wiley-IEEE Computer Society Press, 1996.
[8] J. R. Allen, K. Kennedy, C. Porterfield, J. Warren. Conversion of control dependence to data dependence. *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1983.
[9] C. Wolinski, M. Gokhale, K. McCabe. *Polymorphous Fabric-based Systems: Model, Tools, Applications*, Elsevier Science, 2003.
[10] M. Leeser, P. Belanovic, M. Estlick, M. Gokhale, J. Szymanski, J. Theiler. Applying reconfigurable hardware to the analysis of multispectral and hyperspectral imagery. *Proceedings SPIE* 4480, 2001.
[11] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, E. M. C. Filho. Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers* 49(5), 2000.
[12] T. G. Mattson, B. A. Sanders, B. Massingill. *Patterns for Parallel Programming*, Addison-Wesley, 2004.
[13] C. A. R. Hoare. *Communicating Sequential Processes*, Prentice-Hall, 1985.
[14] M. B. Gokhale, J. M. Stone, J. Arnold, M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. *IEEE International Symposium on FPGAs for Custom Computing Machines*, April 2000.
[15] M. Gokhale, J. Frigo, C. Ahrens, J. L. Tripp, R. Minnich. Monte Carlo radiative heat transfer simulation on a reconfigurable computer: An evaluation. *Proceedings Field-Programmable Logic and Applications (FPL)*, 2004.

RECONFIGURABLE COMPUTING
THE THEORY AND PRACTICE OF FPGA-BASED COMPUTATION

Edited by

Scott Hauck and André DeHon

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW DELHI • NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
Morgan Kaufmann Publishers is an imprint of Elsevier

MORGAN KAUFMANN

Reconfigurable Computing
Hauck and DeHon

MORGAN KAUFMANN PUBLISHERS
An imprint of Elsevier
30 Corporate Drive, Suite 400, Burlington, MA 01803-4255

Original ISBN: 978-0-12-370522-8

First Printed in India 2011

Indian Reprint ISBN: 978-93-80931-86-9

This edition has been authorized by Elsevier for sale in the following countries: India, Pakistan, Nepal, Sri Lanka and Bangladesh. Sale and purchase of this book outside these countries is not authorized and is illegal.

Published by Elsevier, a division of Reed Elsevier India Private Limited.

Registered Office: 622, Indraprakash Building, 21 Barakhamba Road, New Delhi-110 001.
Corporate Office: 14th floor, Building No. 10B, DLF Cyber City Phase-II, Gurgaon-122 002, Haryana, India.

Printed and bound in India by Sanat Printers, Kundli-131 028

CHAPTER 11

OPERATING SYSTEM SUPPORT FOR RECONFIGURABLE COMPUTING

Katherine Compton
Department of Electrical and Computer Engineering
University of Wisconsin–Madison
André DeHon
Department of Electrical and Systems Engineering
University of Pennsylvania

As part of the evolution of the field of reconfigurable computing, researchers are increasingly focusing their attention on the issues of integrating reconfigurable computing into multipurpose or general-purpose compute environments. Operating systems (OSs) fill two key roles in computing: simplifying the programming interface through an abstracted programming model and managing shared resources [40]. Both are critical to reconfigurable computing systems, which have in the past suffered from the stigma of programming difficulty as well as from a general focus on single-application systems and nonscalable, nonportable designs.

An operating system, coupled with the proper compilation environment, can simplify the programming of reconfigurable computing systems by providing a well-defined, well-documented compute model that abstracts the structure and capacity of the underlying hardware. This model may explicitly provide constructs for defining hardware tasks (the parts of the application implemented in reconfigurable logic). Alternately, it may be agnostic to the implementation medium. Like the compute fabric, the communication structures between tasks can be abstracted by the compute model to simplify the design process.

Reconfigurable hardware in a reconfigurable computing system is explicitly intended to be a shared resource. Even in a single-application system, hardware may be shared within the application to accelerate different tasks at different times. In a multitasking system, different threads of computation may vie for the hardware resources. The operating system arbitrates hardware use both within and across applications. Furthermore, the OS also provides protection and security to prevent a maliciously or poorly programmed application from compromising the system. Through isolation, the operating system also provides a safe environment where applications can be debugged and inspected without concern that buggy code will affect system stability.

The demands on an operating system for reconfigurable computing include

- Abstraction of the capacity and composition of reconfigurable hardware resources.
- Scheduling use of shared resources across processes.
- Methods for communication and synchronization among hardware tasks and software.
- Protection of the tasks of one process (hardware and software) from those of another.

This chapter discusses the above concepts in terms of both key roles of an operating system: the programmer's view and the management of shared resources.

11.1 HISTORY

Although the concept of operating system support for reconfigurable computing has existed since at least 1996 [6], the idea languished for a time, not quite gaining popular momentum. A significant barrier to operating system development for reconfigurable computing has been the lack of a standard reconfigurable computing hardware platform as a focus for commercial and academic development.

With much of reconfigurable computing research focused on specialized scientific computers or embedded systems, researchers were willing to forgo the abstraction/virtualization benefits provided by an operating system. Instead, application designers (who frequently were the hardware/system designers) would include hardware management operations in their application, explicitly deciding when and where to load particular operations. Manual management leveraged the designer's understanding of the application to provide potentially better performance than an OS layer, discouraging many researchers from dedicating valuable research time to finding a more generic (but possibly less optimized) solution. Yet these systems too would benefit from operating system support to attract a broader group of application designers uninterested in every hardware detail or in micromanaging its use. Even those with suitable hardware backgrounds could then focus their efforts on application (instead of hardware) details.

The increase in demand for operating system support is mirrored, in part, by the increase in complexity of embedded systems and applications. Many single-function devices of the past have evolved into multifunction devices. Cell phones, for example, not only provide basic voice communication, but also capture pictures and video, replay video and audio, browse the Internet, communicate with other electronic devices, and support gaming. A device may execute several of these applications over time, giving it a "general-purpose" flavor within an "embedded" body. Reconfigurable hardware is attractive for devices such as this because of its flexibility to reconfigure to accelerate a variety of applications. The compute-intensive computations of an application execute

in hardware to operate faster, using less power (battery) than even an embedded instruction-based processor [34, 41].

Even a single-function device may require many different compute-intensive operations. For example, a digital audio player may need to perform error checking, Huffman decoding, IDCT, and other tasks. The reconfigurable hardware that accelerates these operations may, because of cost considerations, be too small to fit all hardware tasks simultaneously. However, an operating system can automatically reconfigure it to implement each task in sequence, as shown in Figure 11.1 (and discussed in Chapters 4, 5, and 9), allowing applications to execute all hardware tasks as if they were persistent in hardware. This provides the application programmer with a virtualized hardware view not hampered by low-level details.

Another contributor to the growing demand for OS support for reconfigurable computing is the increasing difficulty of providing clock speed increases to general-purpose processors [1]. This problem is causing researchers to more closely investigate the potential benefit of reconfigurable computing in general-purpose computers in order to boost performance for compute-intensive applications, including multimedia and communications applications. Using reconfigurable computing in a general-purpose machine requires more

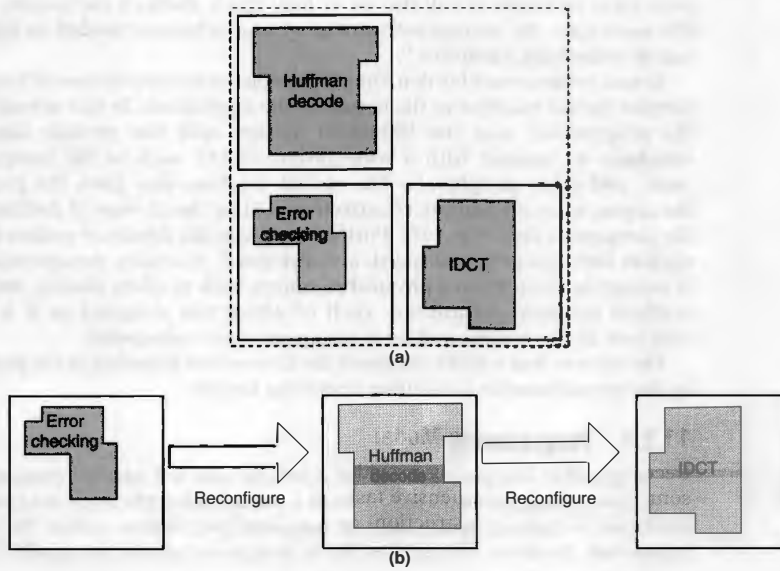


FIGURE 11.1 ■ The abstraction of a large virtual hardware capacity (a) can be implemented on more limited hardware resources using runtime reconfiguration (b).

sophisticated resource management than can be expected from individual applications, further driving the need for OS support.

11.2 ABSTRACTED HARDWARE RESOURCES

The official Commodore *Programmer's Reference Guide* for the C64 computer, originally published in 1982, provides programmers with a great deal of information [10]. For example, it contains a table of the memory map of the C64, including the memory location of the BASIC ROM, the memory-mapped screen output, other memory-mapped I/O, and available program memory locations. It even provides the pinouts of the C64's I/O ports and a schematic of the motherboard—as information to a *programmer*. Much like the preceding evolution of abstracted programming models for mainframe computers [28], increased complexity in personal computing systems later both enabled and required an increase in abstraction.

Today's programming texts do not provide explicit hardware details; instead, for example, they instruct on the use of system calls to provide I/O. One can write an application in a high-level language such as Java or C without knowing even what processor it will run on or how much memory the system will have. For some time, the average software programmer has not needed an understanding of underlying hardware.[1]

To ease programmer burden, the OS provides an abstracted view of hardware—a simpler *virtual* machine as the target for the application. In this virtual machine, the programmer may use library or system calls that provide standardized interfaces to interact with a wide variety of I/O, such as the screen, storage units, and other peripherals. The virtual machine also gives the programmer the appearance of isolation, effectively providing the illusion of dedicated use of the computer's resources [47]. Furthermore, specific details of system resources, such as their quantity and speed, are abstracted. In reality, the operating system is managing these limited physical resources both to allow sharing and to avoid conflicts between applications, each of which was designed as if it were the only one in the system and as if resources were unbounded.

The section that follows discusses the abstraction provided to the programmer by the reconfigurable computing operating system.

11.2.1 Programming Model

Reconfigurable computing provides a mechanism for parallel computation. In some cases, compute-intensive tasks in a sequential application are converted to hardware to capture instruction- or data-level parallelism within the sequential framework. In others, the application is designed explicitly for parallel execution

[1] Embedded systems, some graphics, and other specialized programmers may still require some knowledge of hardware to target specific customized architectures or to meet stringent performance requirements.

throughout, with many concurrent hardware (and software) tasks. Chapter 5, Section 5.1, discusses a variety of compute models for reconfigurable computing.

The application developer, the compiler, or the runtime environment can create multiple interchangeable implementations for a task using a separable interface and implementation. This separation is analogous to the delineation between interface and architecture in VHDL (Chapter 6), or the interface and implementation in Java or C++. The operating system can use the interchangeable implementations to bind the computation to a specific resource at runtime, as discussed in Section 11.3. Compiled applications may be a combination of software components and either abstracted hardware components (which undergo the final steps of compilation/synthesis at install time or runtime) or configuration bitstreams that represent hardware tasks.

Depending on the development environment, designers may explicitly partition their application between hardware and software components, or the compiler may automatically partition a high-level application description (Chapter 26). If explicitly partitioned, the hardware components may be specified in a hardware description language (HDL) or in a high-level language with added constructs to specify parallelism, communication, variable bit width, or other hardware-specific features (e.g., Chapter 7).

Implicit partitioning facilitates application portability, as added language constructs for explicit partitioning may not be available for different systems, and hardware descriptions, while more portable than postsynthesis designs, may still depend on specific hardware features. Automatic partitioning and synthesis at compilation time allows an application description to be easily recompiled for different systems (provided that tool support is available).

Because software programmers are not usually hardware designers, and automatic compilation from a high-level language to hardware does not always provide acceptable results, reusable libraries can provide a balance between ease of specification and result quality. Developers can use library calls to perform compute-intensive operations without concerning themselves with how the operation is actually implemented (hardware versus software, hardware and software details). Libraries can contain efficient hardware implementations, potentially at multiple area/performance tradeoff points, and, possibly, software alternatives for a set of related operations [29, 45]. Static linking to such a library could significantly increase application distribution size if multiple implementation options were included to support different execution platforms or to provide runtime binding (as discussed in Section 11.3). A dynamically linked library (DLL) could ameliorate this problem if it were reused by other applications.

A final approach is to use description languages designed to be agnostic to the eventual implementation in hardware, software, or a mix of the two [13, 22]. Much of the automatic partitioning work focuses on high-level languages normally used in software programming, which were created for inherently sequential compute structures (instruction-based processors). Depending on the hardware design, a reconfigurable computing platform has the potential to provide much more parallelism at a variety of levels difficult to describe using

a software-centric approach. (These concepts are discussed in more detail in Chapter 5.)

Within an application, the programmer or compiler instantiates a hardware task as a virtual resource and later applies it to the suitable input data. When the operating system scheduler (Section 11.4) decides to allocate hardware to the task, it loads that task onto hardware. For best performance, a single hardware task can (and should) execute repeatedly on successive input data. Depending on the extent of runtime support, the operating system could instantiate multiple copies of the task to increase captured parallelism or time-multiplex multiple tasks if hardware resources are limited, as discussed in Section 11.3. This detail should be abstracted from the user, however, as the amount of resources available for the task can be based on runtime system state, which is likely unknown at design time. One approach (discussed in Chapter 9) is to design the application for *maximum possible* parallelism, with the operating system automatically time-multiplexing the different tasks if insufficient resources are available for the full application simultaneously [13].

11.3 FLEXIBLE BINDING

Because reconfigurable computing systems are inherently flexible, they allow the operating system greater freedom in managing shared resources. The operating system can perform flexible binding of tasks to different types of resources (hardware/software) and, for those bound to hardware, can perform a runtime tradeoff between resource use and performance. Flexible binding allows a single application to be implemented using different resources on different computing platforms, or even on the same platform at different times. Install-time binding decisions are based on the physical characteristics of the system (e.g., the number of programmable resources or memories). Runtime binding, on the other hand, uses information about the physical characteristics along with the current system state (e.g., number of running tasks) to make implementation decisions.

11.3.1 Install Time Binding

Install time binding involves the compilation of applications to a generic representation analogous to an intermediate representation in software compilation. Final synthesis of the generic representation occurs at install time based on the specific resource types available on the system. Install time binding is therefore important to the prevailing economic model of computer purchasing: Spending more money does not (generally) allow one to run *different* applications but rather the *same* applications *better*. Likewise, reconfigurable computing machines should be available at multiple price points, with the capacity/performance/power efficiency of their resources increasing in relation to cost.

Applications running on a more expensive, more powerful machine should perform better than those running on a base machine—but they should still *run* on that base machine. In keeping with this economic discussion, if the reconfigurable hardware in a computer is upgraded, the applications may require reinstallation to leverage the new resources. Depending on the level of abstraction of the specification, this may require CAD processing, which should be performed quickly (and potentially in the background when the system is idle) to avoid system slowdown, as discussed in Section 11.3.3 and Chapter 20. An alternate form of install time binding is dynamic linking to precompiled libraries of hardware (and software) task implementations [29]. Libraries can be compiled for different platforms and distributed with the OS as part of the hardware drivers.

11.3.2 Runtime Binding

Runtime binding is based on both physical characteristics and current system state, and may be performed as part of the scheduling process (Section 11.4). It modifies a task's implementation based on the resources allocated to it during scheduling. The most simple form of runtime binding supports relocation of hardware tasks to different regions of the hardware resources. Relocation (discussed in more detail in Chapter 4) facilitates concurrent residency and/or operation of multiple hardware tasks. It also affects task communication, discussed in Section 11.5.

Another form of runtime binding allows a given task to execute in either hardware or software depending on scheduling decisions [14, 29, 31], discussed in Section 11.4.3. Systems that permit dynamic binding can avoid stalling for hardware availability by proceeding with a software alternative for the task. Dynamic hardware/software binding at runtime requires either a task executable capable of running on hardware or software (e.g., [22]) or a pair of interchangeable hardware and software implementations [13, 14, 29]. To facilitate application design and debug, the two components should have identical functional behavior.

Runtime binding can allow hardware tasks to expand or contract to make use of the resources allocated to them by the scheduler, as discussed in Section 11.4. This ability allows tasks to be implemented on a variety of architectures, from low capacity to high capacity, to promote portability. Hardware tasks can also be modified based on system load, occupying fewer resources in a system under heavy load and more in a system under light load, as shown in Figure 11.2. In (a), task A is using fewer resources because of increased demand by other tasks. In (b), task A rebounds to more resources after task B is no longer needed. Task A's data rate is improved in (b) by the increased parallelism.

A task can occupy fewer resources by time-multiplexing its functionality, or more resources by unrolling or replicating [13]. Time-multiplexing a task requires storage to hold intermediate results between the temporal partitions. Performing time-multiplexing or expansion at runtime can be quite expensive, potentially involving a modified CAD flow, as discussed next in Section 11.3.3. Alternately, implementations at multiple area–performance (or power) tradeoffs can be created at compilation time, eliminating transformation overhead at runtime [14, 29].

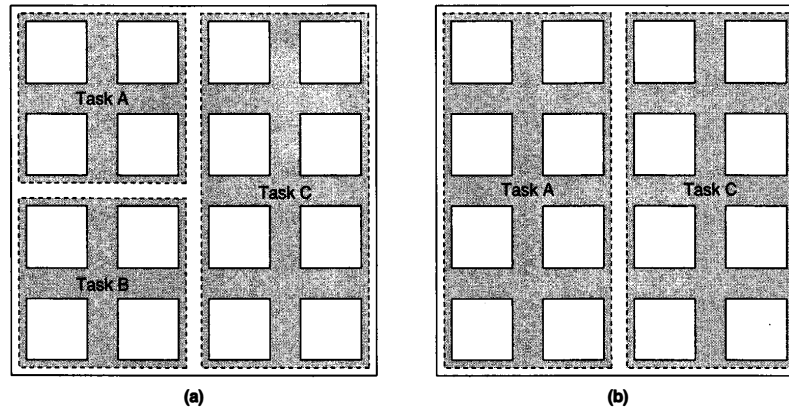


FIGURE 11.2 ■ Flexible binding allows tasks to use a different number of resources based on hardware capacity and resource availability. In this example, Task A can either occupy less area at the expense of performance (a), or achieve a higher data rate at the expense of area (b).

Although a specific palette of implementations reduces OS complexity, it also limits the possibilities of customizing the hardware task to the exact hardware resources available.

11.3.3 Fast CAD for Flexible Binding

Modifying a hardware task after application distribution may require that one or more CAD operations, such as placement, be applied at install time or runtime [13, 39, 43, 44] (e.g., Section 9.4). Unfortunately, CAD algorithms, depending on the problem size, can be quite slow. Chapter 20 discusses a number of fast CAD approaches for hardware task implementation motivated in part by flexible binding. Some possible solutions to accelerating install time or runtime CAD processes include

- Trading solution quality for speed in the CAD process (less optimized solutions).
- Accelerating CAD algorithms in hardware (i.e., implementing CAD hardware tasks on the target reconfigurable computing system).
- Abstracting some of the hardware detail to simplify the problem (applying algorithms to larger blocks of structures, where intragroup CAD decisions are fixed at compile time, and only intergroup CAD decisions are required at install time or runtime, as discussed in Chapter 4 and Section 9.2.4).
- Using a compile time CAD process to generate static information about the hardware task that can be used to accelerate later CAD operations (marking areas of the circuit for replication or time-multiplexing).

11.4 SCHEDULING

Scheduling determines what tasks should use hardware when, and may also decide how many resources (and what type) to allocate to each. These decisions may be made at compile time based on static application information, at runtime based on dynamic system status, or at a combination of the two. The scheduling goals may include maximizing application or system performance, minimizing power consumption, or meeting real-time deadlines. Achieving these goals also requires minimizing the reconfiguration overhead, as discussed in Chapter 4.

Schedulers that include resource allocation also perform flexible binding (Section 11.3), choosing specific resources to implement a given task and potentially altering that task to fit the resources. Flexible binding complicates the scheduler's decision process by expanding the search space. However, expanding the search space with flexible binding also opens the door to scheduling solutions that would otherwise not be possible.

11.4.1 On-demand Scheduling

One of the simplest forms of runtime scheduling is servicing hardware resource requests in the order received, reconfiguring as needed, and queuing requests that cannot yet be serviced [6]. When an application calls a hardware task, its request is sent to the operating system. If the task is preconfigured on hardware, it executes; otherwise, it must be loaded into hardware (configured) prior to execution. If all hardware resources are allocated and in use, the system will queue waiting requests until the resources are freed.

Hardware requests are generally blocking, forcing the requestor to busy-wait until the hardware is available. Then the task is configured and finally executes. Busy-waiting can contribute significantly to reconfigurable computing overhead, as discussed in Chapter 4, but the system (with an appropriate compute model) can use a sleep/wake approach instead of busy-waiting to allow nonblocking threads or processes to use the compute resources in the meantime, hiding some of the configuration latency. Furthermore, runtime binding, discussed in Section 11.3, allows threads or processes that might otherwise be blocked waiting for hardware availability to execute in software instead.

11.4.2 Static Scheduling

Static scheduling relies on analyzed, profiled, or annotated application behavior to determine when an application should request that each hardware task be configured [23, 26, 27]. Static schedulers operate "offline" and thus have a more global view of the task requirements and are able to search a greater expanse of the solution space than a dynamic (online) scheduler. Brute-force or Monte Carlo approaches may therefore be feasible for static schedulers even if prohibitively slow for dynamic scheduling. A static scheduler can also attempt to load hardware tasks prior to their execution to minimize configuration overhead (a technique known as prefetching [24]).

For static scheduling to be profitable, however, both the application task set and resource availability must be highly predictable. An offline schedule does not have access to runtime information and therefore cannot adapt to the current system load. This can prevent the static schedule from computing a good coschedule of multiple independent tasks. Further, if the static schedule is wrong about which tasks must run next, prefetching can actually be detrimental to performance, forcing needed configurations to be evicted and performing extra, unnecessary reconfigurations.

11.4.3 Dynamic Scheduling

Dynamic schedulers use runtime information to aid scheduling. Data-dependent application behavior, system load, and the characteristics of other executing applications can therefore all contribute to (and complicate) schedule computation. Although single-application behavior may be statically predictable in some cases, the interferences arising from multiple simultaneously executing applications lead to an explicitly nondeterministic interleaving of hardware task calls from different applications.

As a simplification, some schedulers use a window-based approach, dividing time into windows and solving the scheduling problem for each [14, 27, 31]. Figure 11.3 illustrates the timing of window-based scheduling. Once the scheduler determines which tasks should be implemented in hardware, the hardware must be reconfigured to implement them. After reconfiguration, the hardware can execute until the next reconfiguration phase in the following window. To minimize the impact of scheduling overhead, the window should be "large" compared to the time required to compute the schedule and perform reconfiguration. However, it should also be small enough to capture current system behavior for use in the scheduling decision. Statistics from the previous interval (or multiple previous intervals) provide recent behavior information to the scheduler.

A "frontier" dynamic scheduler [27] uses application dataflow and task execution information from the previous interval (such as which tasks executed and at what data rate) to compute the new schedule for the next interval. Input data availability and allocatable output space information are requirements for task scheduling and are used to compute the relative priority of tasks. The scheduler can use resource availability and information on data rate (of the considered task

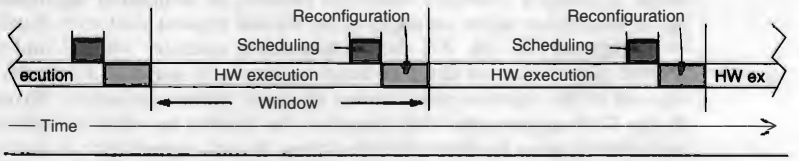


FIGURE 11.3 ■ A dynamic scheduler can divide time into a series of windows, each with its own scheduling problem.

and ones it communicates with) to choose a flexible binding implementation. This approach matches a streaming communication approach (Chapter 9), where good scheduling is necessary to minimize the buffering requirements between tasks. Flexible binding allows the frontier scheduler to time-multiplex or replicate tasks to balance the data rate between one task and those adjacent [27].

Runtime information, such as the frequency of task use in the prior interval and task performance information, also can be used without considering the dataflow of the executing applications. In each window, hardware resources can be treated as a knapsack, which the scheduler tries to pack with the greatest overall value [14]. Each task is assigned a "value" based on performance or power consumption and a "cost" based on hardware area requirements. Tasks not scheduled to hardware execute with lower performance in software to avoid starving less valued tasks. By including multiple implementations of a task with different values/costs, the scheduler can also use dynamic binding to adapt task implementations based on resource availability/demand [14, 27]. The knapsack problem can be solved either heuristically or, if the problem size is small enough, exactly.

11.4.4 Quasi-static Scheduling

A purely dynamic scheduler only considers information available at runtime and loses the opportunity to optimize based on known application characteristics. In contrast, quasi-static scheduling combines dynamic system and application information with static application analysis. Using dynamic management with static analysis enables the scheduler to more accurately predict near-future hardware task needs (and, just as important, which tasks will *not* be needed) [24, 27]. Quasi-static scheduling also accelerates the scheduling process by reducing the dynamic scheduler's burden.

For example, static analysis can provide the ordering of tasks within an application, timing estimates for when the tasks will be executed relative to one another, data rate analysis of different possible time-multiplexing/replications of the tasks, and intertask communication resource requirements. The runtime scheduler can then use dynamic scheduling techniques, but prune the solution space based on static analysis information to arrive at an improved solution more quickly. Dynamic scheduling can also allow otherwise statically scheduled applications to reuse hardware tasks configured for other applications to reduce configuration costs [35].

11.4.5 Real-time Scheduling

Scheduling for real-time systems considers task deadlines rather than general performance. Hard deadlines must be met within the specified time or the system has failed. An example of a hard deadline would be triggering operation of strictly timed automotive engine components. Soft deadlines must be generally met for acceptable use, but missing one or even a few is not mission-critical. An example of missing a soft deadline would be dropping a frame in real-time video.

Missing a soft deadline may not invalidate the computation, but it may degrade the application in some way. This type of operation is common in embedded systems. Indeed, real-time systems and their operating systems are the focus of much research [20, 23].

One approach to implementing reconfigurable real-time systems is to leverage the vast real-time research effort by wrapping hardware tasks with a thread interface [4]. Such a system includes a generic hardware-based scheduler for both hardware and software threads using whatever scheduling algorithm is implemented within it. Synchronization details of this approach are presented in Section 11.6.1.

Alternately, the scheduling algorithm can be tailored specifically to reconfigurable computing, using information about hardware capacity, task hardware requirements, and task configuration time in addition to deadline information [39, 43]. For example, tasks that can fit in a currently available area are more likely to be guaranteed to meet a deadline than are those that require reconfiguration due to reconfiguration overhead. If sufficient resources are free, but are distributed throughout the hardware, defragmentation may be required (see Chapter 4) to consolidate sufficient free space for the incoming task.

The time required for this process affects the ability of the system to meet the task's deadline. If free space is not available even with defragmentation, the task may be rejected or its deadline not guaranteed. The task could meet the deadline if one or more other tasks executing on hardware complete with enough time left to permit configuration and execution of the new task before its deadline expires. Alternately, a task implemented in hardware may be preempted (see next section) in favor of an incoming task if the latter has higher priority [43].

11.4.6 Preemption

A scheduler may use preemption to reallocate hardware to a "more desirable" task, whether based on meeting specific deadlines in a real-time system, based on the relative priority of different tasks in a performance-based system, or to allow a more balanced use of hardware in the presence of long-executing tasks [2, 18, 31, 43]. The configuration data for a given task holds some of the required information, such as circuit structure, and possibly initial values for embedded memories. However, any values in state-holding elements that change in response to hardware operation are not included. Therefore, the complete "saved state" for preempting a hardware task is a combination of its configuration data and the current values of any state-holding elements modified during execution. Provided a hardware interface to this information is available, the operating system can read the current state to store in memory and later load it back into hardware when needed.

Preemption is complicated by flexible binding if the implementation saved does not match the implementation resumed. The sizes of configuration data and the number of state-holding elements may not match between different implementations. Therefore, systems supporting flexible binding and preemption must save an abstracted view of task state.

11.5 COMMUNICATION

A key feature of communication abstractions is that they are, in fact, abstractions. Although certain abstractions may map well to specific hardware architectures (and vice versa), the use of one in particular does not necessarily require a particular hardware structure (or vice versa). For example, a message-passing abstraction could be implemented on a shared memory architecture, or a shared memory abstraction could be implemented on top of a message-passing architecture. Library calls and the compilation environment map communication abstractions to the actual implementation, and the operating system manages the implementation. The abstractions, however, allow the programmer to ignore implementation details and focus on efficient specification. The following subsections discuss a number of abstractions and their operating system requirements, along with other communication issues requiring operating system intervention.

11.5.1 Communication Styles

When our applications are composed from multiple, concurrent tasks (e.g., threads, hardware tasks, software tasks, operators), the tasks must often exchange intermediate data in order to solve the entire problem. Specifying this communication can be highly error prone and performance critical. The form in which the communication is specified should match both the natural compute model (see Section 5.1) for the application and the nature of the communication required.

Shared memory

Shared memory is an implicit form of communication motivated by certain implementations where tasks share a common memory pool (Single memory pool subsection of Section 5.1.4) and address space, or share a mapped portion of an address space (Section 11.5.2). Here, the semantics are that each task sees the same image of memory. If one task writes to the image, another should be able to see the values written to the memory. In this way, the memory addresses serve as named locations through which values can be exchanged among tasks.

Uniprocessor operating system developers see shared memory as a particularly efficient way of communicating between tasks. In multithreaded environments where tasks are interleaved in time on the same processor, shared memory segments within the single memory hierarchy allow multiple tasks to share data without an explicit need for data to be copied between the routines. This can minimize the overhead for data communication between tasks. Without caches, shared bus multiprocessing systems with a common main memory would exhibit a similar efficiency. Local caches potentially complicate the picture. However, good architecture and engineering can maintain this abstraction efficiently in the common case, at the cost of additional hardware to support cache coherence.

A reconfigurable computing architecture may nevertheless more closely mirror, at the chip or board level, the organization of a large, distributed memory system.

Here, data may actually need to be copied between distant memories, complicating the shared memory abstraction. The result is both significant hardware overhead to support the model and, often, significant communication time overhead beyond what would be required to move the data between the producer and the consumer. Furthermore, synchronization between shared memory threads/tasks (Section 11.6.1) is a common source of application errors, leading some to question the viability of this model for capturing larger-scale parallelism [21, 36].

Method calls

As the previous section suggested, word-level shared memory is a very low-level form of implicit communication that is prone to synchronization issues. Implementing the abstraction can increase hardware requirements in architectures containing distributed memories. In modern object-oriented systems, particularly when each object may itself be an independent thread, a higher-level communication technique is method calls between objects or operators (see Section 5.1.2). The method call on the object explicitly states the intended destination for the data; further, the object method provides additional semantic information to the receiver about what the data means. As long as object methods are serialized on each object, method invocation can be atomic, providing a natural mechanism for consistent updates to object state. In some cases, method calls can eliminate the need for a hardware task to communicate directly with memory, allowing many lightweight, reconfigurable operators to avoid the expense of a memory interface unit.

When the destination object is running on hardware that is physically distinct from the sending object, the method invocation, and the communication in general, requires data to be routed from the sending to the receiving hardware. This is true even in a shared memory implementation—the method call style simply makes this communication explicit. However, when the objects share the same physical memory, method call communication can still occur through shared memory.

Message passing (discussed in the Message passing subsection of Section 5.2.6) is a form of method call communication, as is remote procedure call [30]. MPI [38] is a well-developed standard for message passing, and reconfigurable computers have been built to interface with standard MPI communications [32]. However, MPI itself is fairly heavyweight, and its overhead may be too high for finer-grained composition of tasks and operators. Lighter-weight message passing designed for on-chip reconfigurable applications has been developed [31], as have remote procedure call interfaces for symmetric use between processors and reconfigurable logic [8].

Streams

While method invocation is an explicit communication mechanism, it is still dynamic and does not provide the OS with advanced warning about which tasks will communicate and when. Further, the actual graph of communication remains implicit in the object call structure. A more explicit form of

communication is to represent the graph structure for task communications and share that information with the operating system. This is similar to the use of pipes or streams in conventional software multi-threading to represent persistent communication links between communicating threads. The reconfigurable computing dataflow models in Section 5.1.3, and the streaming dataflow programming approaches in Chapters 8 and 9 provide some ways to capture these communication graphs. Data-centric compute models (Section 5.1.6) do so as well.

Streams (pipes, channels) are persistent, unidirectional links between tasks (software or hardware) that pass data or control information. Tasks receive available data from one or more input streams and write the results of their computation to one or more output streams [9, 13]. A stream may buffer data in a FIFO manner between the producer and consumer to allow them to run independently of each other and minimize the effects of both reconfiguration and communication latency. Figure 11.4 is an example that shows abstract use of streams (a) and its implementation on a streaming architecture (b). Sections 5.1.3 and 5.2.1 and Chapter 9 present in-depth discussions of streaming models and architectures.

Because the structure of communication (producer–consumer) is explicit, the operating system is able to more easily make intelligent decisions about where to place tasks to promote physical locality, and the scheduler is able to better choose when to run them. For example, if a stream between a producer and consumer is empty or near empty, the scheduler knows that it is more profitable to run the producer than the consumer. A very full stream would imply the opposite.

The persistence of abstract streams allows us to separate the part of communication that specifies the location (source/destination) of data from the part

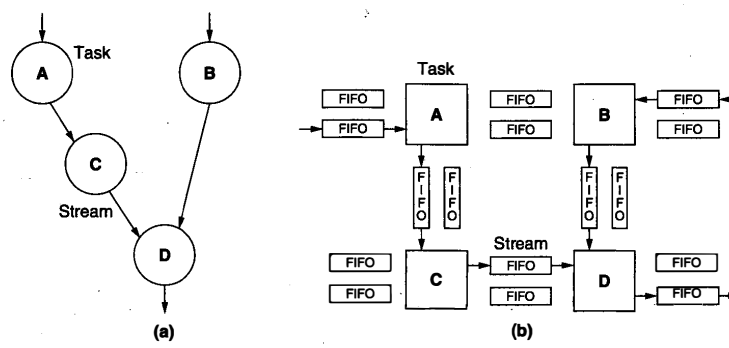


FIGURE 11.4 ■ A stream abstraction defines application dataflow (a); a streaming architecture can implement the streams between tasks using FIFOs (b).

that provides or uses it. For regular communications, this brings the destination specification out of the inner loop of communication, reducing communication overhead. For spatial, reconfigurable datapaths, it allows a stronger correlation between the abstraction and implementation of communication between currently-executing hardware tasks, reducing overhead. The stream can be implemented with simple wires, or a FIFO, between the producer and consumer. Nonetheless, although specifying, allocating, and setting up the stream can be expensive, for heavily used, persistent communications, the long use over time amortizes the cost of stream setup. Short communication sequences or communications to short-lived tasks may not be able to amortize this cost and may be better served with a different communication scheme.

Stream abstraction can be implemented efficiently on a variety of physical communication structures. It can be supported efficiently on a shared memory system with the use of a well-designed and well-tested queue object library that encapsulates the explicit synchronization necessary to implement the stream. Encapsulation is a huge benefit in that it allows one highly trained system programmer to work out a robust locking discipline that can then be used by other programmers with less (or no) experience with synchronization primitives. Stream data can be packed into efficient, longer messages on packet-switched, message-passing systems, or it can be supported by concurrent direct memory access (DMA) data transfers. A message-passing implementation of a stream abstraction can also be extended across the Internet using TCP/IP connections. As noted earlier in this section and elaborated in Chapter 9, when the source and the sink are coresident, the stream can reduce to a direct, configured connection between tasks, requiring minimum hardware and latency overhead during operation.

11.5.2 Virtual Memory

Software applications for general-purpose systems use a virtual memory abstraction, enabled by a combination of hardware and software, to simplify the programming model and to provide isolation (protection) from other processes. Reconfigurable computing systems require this abstraction for the same reasons.

To avoid the complexities of virtual address translation in reconfigurable hardware, the reconfigurable computing system designer may place the burden of memory communication on host processor resources, which already support virtual memory. When the reconfigurable unit is tightly coupled with a processor, it can explicitly share the processor's memory management unit (MMU) [18]. Alternately, the processor can perform memory accesses for a hardware task, feeding data to the task through a dedicated buffer structure [15]. The drawback of using the processor in this fashion is a lack of efficiency. The processor is consigned to acting as an overqualified memory controller, which reduces its availability for parallel computation.

To leverage the processor's address translation capability (including translation lookaside buffer [TLB] miss processing and page fault handling) and at the same time remove the processor from the inner memory access loop, a DMA-style

approach can be used. The processor provides hardware with translated physical addresses for the needed virtual addresses. User hardware should not, however, be able to issue these accesses directly, as it could potentially issue memory requests to other physical addresses outside the task's virtual memory space. An architectural solution to this problem is to add one or more hardware memory address generators that are guaranteed to abide by the virtual memory abstraction. The address generator may require the processor to translate all addresses, or it potentially can combine offsets from the hardware task with a translated page or segment base address to further reduce processor involvement.

Finally, a dedicated hardware MMU can directly translate virtual addresses to physical ones [16, 42]. It maintains its own copy of the TLB for address lookups. TLB misses can be handled either by the hardware MMU itself or by interrupting a processor to walk the page table. In this arrangement, page faults are handled by the operating system, which updates the hardware MMU's TLB based on the result.

11.5.3 I/O

Finally, in addition to communicating with other tasks, a hardware task may need to communicate with system I/O. Libraries abstract the hardware interfaces for the programmer [11] (as discussed in Chapter 8). However, I/O standards are continually evolving and can do so during the lifetime of a given application. The operating system, through I/O device drivers, can support changing I/O standards by providing these libraries in dynamically linked form so that they can be updated and expanded without requiring any changes to the applications in order to use them.

11.5.4 Uncertain Communication Latency

Communication between tasks (and memory) is subject to uncertain latencies for a number of reasons. One common example in many traditional computing systems is the uncertain latency of memory access due to location in the memory hierarchy and memory contention. Reconfigurable computing systems share this problem. However, those that support flexible binding (Section 11.3) are subject to additional sources of uncertainty, as different implementations of a given task have different data rates. Even given the same implementation of a hardware task, its location on hardware can affect the latency of communication between it and other tasks. Depending on the physical implementation of the routing network between physical task locations, some locations may be "closer" than others.

Although this could create variable clock rates depending on task locations, the problem is easily addressed using pipelined interconnect and data presence (discussed in the Data presence subsection of Section 5.2.1 and in Chapter 9). The same set of data presence techniques also support flexible binding where a task implemented in hardware can have a much higher data rate than one implemented in software.

11.6 SYNCHRONIZATION

Reconfigurable computing applications are generally concurrent, executing one or more hardware tasks in parallel along with one or more software tasks. Therefore, they require synchronization between tasks. A number of factors complicate synchronization in reconfigurable computing. First, reconfigurable computing applications can leverage a variety of parallelism types (instruction-level, data-level, task-level, pipeline-level) to a greater degree than software-only applications can, as discussed in Chapter 5. More parallelism exacerbates the already difficult process of concurrent programming [33]. Furthermore, runtime binding and placement can affect communication source/destination locations and task data rate even after program specification and compilation. Given this degree of parallelism and uncertainty, effective synchronization techniques are critical to reconfigurable computing application design and performance.

These effects are mitigated to some extent by the fact that reconfigurable computations and data often use distinct resources with less potential sharing; this can often clarify the synchronization required and permits more coarse-grained resource locking. Depending on the abstraction employed, synchronization may be controlled explicitly by the programmer or implicitly by the operating system or underlying hardware.

11.6.1 Explicit Synchronization

Synchronization between tasks can be performed explicitly through abstractions similar or identical to those used in software-only multi-threaded programming. This approach is particularly appealing in embedded systems, where application designers may have used a shared memory multi-threaded model more widely than the average general-purpose computer programmer would have. As in software-only shared memory applications, constructs such as locks and semaphores can protect access to shared resources to avoid race conditions.

We can impose thread-style interfaces on hardware tasks [4, 7, 42]. The thread interface requests/releases a semaphore and forces hardware to stall or sleep while waiting to acquire one. Memory structures within the hardware must be augmented with a table to hold semaphore information. This has the advantage of hiding details of the hardware task implementation from the communicating thread but at the cost of logic overhead to interface hardware with the shared memory pool that holds the synchronization address.

11.6.2 Implicit Synchronization

Low-level, thread-style synchronization, already prone to design error and debug difficulty, is likely to become even more difficult to implement correctly as the degree of parallelism required to achieve demanded performance increases [21, 36]. Instead, designers could turn to abstractions that provide more explicit parallelism with implicit synchronization.

To efficiently use our reconfigurable resources, we typically provide them with large blocks of data at a time contained in contiguous memory addresses

(e.g., an image frame). Thus, it is natural to give exclusive ownership of a memory block to a hardware task during its execution. By combining this locking with the instantiation semantics for the operation, we can automate locking to prevent the programmer from having to manage it explicitly. This can even be supported by hardware using a scoreboarding technique similar to the ones used to prevent hazards in aggressive processor pipelines [19].

Synchronization is implicit in all forms of dataflow (Chapter 5, Section 5.1.3). The semantics of its operation are based on data arrival, not sequential timing, which makes proper synchronization the job of the compiler, the hardware, and the runtime system rather than the programmer. In streaming dataflow, stream data comes with data presence information (see Section 11.5.1 and Chapter 9). In general dataflow, I-structures allow fine-grained synchronization and concurrent cooperation on common data structures [5].

11.6.3 Deadlock Prevention

Whether synchronization is implicit or explicit, the need for it in a concurrent application presents the unfortunate opportunity for deadlock. Essentially, one or more tasks in the application may not be able to continue because they are waiting on other tasks. When the waiting set forms a cycle, the system will never be able to make forward progress. However, because deadlock can arise only when a task needs exclusive access to multiple resources simultaneously, many hardware tasks will work on a single, coarse-grained set of data at a time, avoiding this issue. Nonetheless, it is common for a hardware task to need multiple resources (e.g., one or more input buffers and an output buffer).

A common method to prevent deadlock is to force tasks to acquire all of their resources in a canonically ordered sequence. This way we avoid deadlock by never creating a cyclic dependence that could lead to it. With implicit and higher-level locking, runtime support mechanisms can provide the ordering guarantee. This demands that we establish a canonical ordering for all resources that might be locked, both in hardware and in memory locations, and use it uniformly throughout the system.

11.7 PROTECTION

Modern computing systems all share a need for protection from processes (intentionally or unintentionally) interfering with one another. This protection is critical for dealing with not only maliciously coded applications but also poorly programmed ones. During the application development process, isolation is critical because it allows designers to test and inspect their implementations. Development is significantly more complicated if bugs can bring down the development system, destroying state information critical to the debugging process. The same need for protection holds for reconfigurable computing systems. The operating system must prevent processes from

using hardware inappropriately or from interfering with or intercepting communication between tasks (hardware or software) of other processes. Some of these responsibilities fall to the scheduler—preventing task resource starvation is one example; others fall to the hardware allocator (which may be part of the scheduler); still others fall to the system's hardware interface.

11.7.1 Hardware Protection

Implementing user tasks as hardware circuits in the reconfigurable fabric introduces a major security flaw unfathomable to the average software user or developer. Depending on the underlying hardware design, a hardware task can cause a short circuit, permanently damaging the computing system. Therefore, either the hardware structure itself must prevent the possibility of short circuits [3, 46] or the operating system must screen user hardware and prohibit any implementations that cannot be proven to be free of short circuits.

Even if an individual task does not cause a short circuit, incorrectly allocating hardware resources to more than one task can create one. That is why the allocation process must physically separate tasks [44]. Figure 11.5 shows a generic FPGA architecture with resources allocated to two different tasks (separated by the heavy dashed line). Wires shown in bold cross the boundaries between tasks, causing potential conflicts. Resources that cross task boundaries can be allocated to no more than one task unless they are part of intertask communication (discussed in the next section). This restriction also prevents maliciously designed tasks from "snooping" communication paths to which they should not have access (also discussed in the next section).

General FPGA structures complicate the task interference problem by having large numbers of extremely flexible routing structures that may span large distances in the hardware. In contrast, some architectures designed specifically for

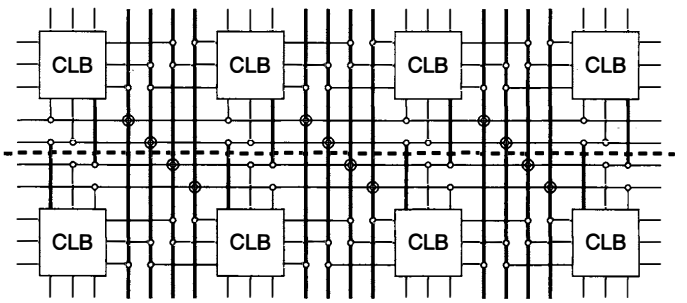


FIGURE 11.5 ■ A generic FPGA architecture may have resources (*bold lines*) that cross the boundary (*dashed line*) between two hardware tasks.

reconfigurable computing, such as SCORE (Chapter 9 and [13]) and PipeRench (Chapter 2, Section 2.1.2, and [17]), are composed of sets of reconfigurable logic (pages/blocks/stages) that are more self-contained, and are the atomic hardware unit for task allocation. Restricted, well-structured connections between these blocks simplify the problem of preventing cross-task interference.

11.7.2 Intertask Communication

As discussed in Section 11.5.2, virtual memory provides each process with a separate address space, preventing one process from accessing the memory space of another. For the same reason, we must provide similar isolation for other forms of communication.

Point-to-point communication, too, can provide isolation if we can guarantee that tasks can only access communication paths owned by their process. The programming model may support this view, but simply trusting it would be equivalent to trusting that compilers will not allow hackers to create viruses. The system (hardware and operating system) must ensure that the isolation model is enforced.

To provide isolation, the system could allow only indirect intertask communication through shared virtual memory [15, 16]. However, this approach can introduce significant communication latencies if both tasks are present in hardware close to one another, but communicate through a relatively distant memory hierarchy acting as intermediary. Safe direct on-hardware intertask communication can be implemented by treating intertask communication routing as special resources that cannot be self-allocated by a hardware task description. Instead, the operating system must allocate these resources when configuring the related tasks onto hardware [13]. By removing user control over allocation of these resources, the isolation the programming model provides is implemented by the operating system. This is much like how only the OS is allowed to manipulate the page tables and TLBs that support the virtual memory abstraction.

11.7.3 Task Configuration Protection

The loading of tasks into hardware must be restricted to the operating system to ensure that the OS has an accurate view of hardware for scheduling/allocation decisions and to enforce hardware and communication protection as discussed previously. Hardware communication paths must therefore be accessible only to OS kernel-level processes. An operating system can isolate task addressability by employing a model akin to virtual memory, where each process can address its own tasks only. Any tables of task information used by the operating system in this case include the process ID as part of the task ID. Any requests for task access are within the user task ID space. Isolation not only prevents processes from triggering the execution, reconfiguring, removing, or altering of tasks from another process, but it also reinforces the abstraction that processes have the hardware to themselves.

11.8 SUMMARY

The primary role of the operating system is to provide abstraction. Abstraction benefits the application designer in the following ways:

- By simplifying the design process to remove the burden of low-level details.
- By allowing the application to run on various hardware platforms and capacities.
- By implementing a virtual machine for each application to prevent interference between them.

This chapter presented the needs, opportunities, benefits, and techniques surrounding the abstraction of reconfigurable resources. It also showed how abstraction affects the application specification process, and discussed the issues involved in implementing these abstractions in the operating system and architecture of the reconfigurable computing system.

References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, D. Buger. Clock rate versus IPC: The end of the road for conventional microarchitectures. *International Conference on Computer Architecture*, 2000.

[2] A. Ahmadinia, C. Bobda, D. Koch, M. Majer, J. Teich. Task scheduling for heterogeneous reconfigurable computers. *Symposium on Integrated Circuits and System Design*, 2004.

[3] R. Amerson, R. Carter, W. Culbertson, P. Kuekes, G. Snider, L. Albertson. Plasma: An FPGA for million gate systems. *ACM International Symposium on Field-Programmable Gate Arrays*, 1996.

[4] D. Andrews, D. Niehaus, R. Jidin. Implementing the thread programming model on hybrid FPGA/CPU computational components. *Workshop on Embedded Processor Architectures, International Symposium on Computer Architecture*, 2004.

[5] Arvind, R. S. Nikhil, K. Pingali. I-Structures: Data structures for parallel computing. *Proceedings of the Workshop on Graph Reduction*, 1986.

[6] G. Brebner. A virtual hardware operating system for the Xilinx XC6200. *International Workshop on Field-Programmable Logic and Applications*, 1996.

[7] G. Brebner. Multithreading for logic-centric systems. *International Conference on Field-Programmable Logic and Applications*, 2002.

[8] M. Budiu, M. Mishra, A. Bharambe, S. C. Goldstein. Peer-to-peer hardware–software interfaces for reconfigurable fabrics. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002.

[9] M. Butts, A. M. Jones, P. Wasson. A structural object programming model, architecture, chip and tools for reconfigurable computing. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2007.

[10] Commodore Business Machines. *Commodore 64: Programmer's Reference Guide*, H. W. Sams, 1982.

[11] C. Chang, J. Wawrzynek, R. W. Brodersen. BEE2: A high-end reconfigurable computing system. *IEEE Design and Test of Computers* 22(2), 2005.

[12] M. Dales. Managing a reconfigurable processor in a general purpose workstation environment. *Design, Automation and Test in Europe*, 2003.
[13] A. DeHon, Y. Markovskiy, E. Caspi, M. Chu, R. Huang, S. Perissakis, L. Pozzi, J. Yeh, J. Wawrzynek. Stream computations organized for reconfigurable execution. *Microprocessors and Microsystems* 30, September 2006.
[14] W. Fu, K. Compton. An execution environment for reconfigurable computing. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005.
[15] W. Fu, K. Compton. A simulation platform for reconfigurable computing research. *International Conference on Field-Programmable Logic and Applications*, August 2006.
[16] P. Garcia, K. Compton. A reconfigurable hardware interface for a modern computing system. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2007.
[17] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, R. Laufer. PipeRench: A coprocessor for streaming multimedia acceleration. *International Symposium on Computer Architecture*, May 1999.
[18] J. R. Hauser. *Augmenting a Microprocessor with Reconfigurable Hardware*, Ph.D. thesis, University of California, Berkeley, 2000.
[19] J. A. Jacob, P. Chow. Memory interfacing and instruction specification for reconfigurable processors. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 1999.
[20] H. Koptez. *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, 1997.
[21] E. Lee. The problem with threads. *Computer* 39(5), May 2006.
[22] B. Levine, H. Schmit. Efficient application representation for HASTE: Hybrid architectures with a single, transformable executable. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003.
[23] Z. Li, K. Compton, S. Hauck. Configuration caching management techniques for reconfigurable computing. *IEEE Symposium on FPGAs for Custom Computing Machines*, 2000.
[24] Z. Li, S. Hauck. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, 2002.
[25] J. W. S. Liu. *Real Time Systems*, Prentice-Hall, 2000.
[26] R. Maestre, F. J. Kurdahi, M. Fernández, R. Hermida, N. Bagherzadeh, H. Singh. A framework for reconfigurable computing: Task scheduling and context management. *IEEE Transactions on VLSI* 9(6), December 2001.
[27] Y. Markovskiy, E. Caspi, R. Huang, J. Yeh, M. Chu, J. Wawrzynek, A. DeHon. Analysis of quasi-static scheduling techniques in a virtualized reconfigurable machine. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2002.
[28] G. H. Mealy. The functional structure of OS/360, Part I: Introductory survey. *IBM Systems Journal* 6(1), 1966.
[29] N. Moore, A. Conti, M. Leeser, L. S. King. Writing portable applications that dynamically bind at run time to reconfigurable hardware. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2007.
[30] B. J. Nelson. *Remote Procedure Call*, Xerox Palo Alto Research Center technical report, 1981.
[31] V. Nollet, P. Coene, D. Verkest, S. Vernalde, R. Lauwereins. Designing an operating system for a heterogeneous reconfigurable SoC. *Proceedings of the Reconfigurable Architectures Workshop*, 2003.

[32] A. Patel, C. A. Madill, M. Saldana, C. Comis, R. Pomes, P. Chow. A scalable FPGA-based multiprocessor. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006.
[33] S. Qadeer, D. Wu. KISS: Keep It Simple and Sequential. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2004.
[34] J. Rabaey. Reconfigurable processing: The solution to low-power programmable DSP. *Proceedings of ICASSP*, April 1997.
[35] J. Resano, D. Mozos, F. Catthoor. A hybrid prefetch scheduling heuristic to minimize at runtime the reconfiguration overhead of dynamically reconfigurable hardware. *Design, Automation, and Test in Europe*, 2005.
[36] S. Singh. Integrating FPGAs in high-performance computing: Programming models for parallel systems—the programmer's perspective. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2007.
[37] G. Snider, B. Shackleford, R. J. Carter. Attacking the semantic gap between application programming languages and configurable hardware. *International Symposium on Field-Programmable Gate Arrays*, 2001.
[38] M. Snir, W. Gropp. *MPI: The Complete Reference*, 2nd ed., MIT Press, 1998.
[39] C. Steiger, H. Walder, M. Platzner. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Transactions on Computers* 53(11), 2004.
[40] A. S. Tanenbaum. *Modern Operating Systems*, Prentice-Hall, 1992.
[41] R. Tessier, W. Burleson. Reconfigurable computing and digital signal processing: A survey. *Journal of VLSI Signal Processing* 28(1–2), 2001.
[42] M. Vuletic, L. Pozzi, P. Hauck. Seamless hardware-software integration in reconfigurable computing systems. *IEEE Design and Test of Computers* 22(2 N), 2005.
[43] H. Walder, M. Platzner. Online scheduling for block-partitioned reconfigurable devices. *Design, Automation and Test in Europe*, 2003.
[44] G. Wigley, D. Kearney. The development of an operating system for reconfigurable computing. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.
[45] M. J. Wirthlin, B. L. Hutchings. A dynamic instruction set computer. *IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.
[46] Xilinx. *XC6200 FPGA Advanced Product Specification*, June 1996.
[47] B. Ylvisaker, B. Van Essen, C. Ebeling. A type architecture for hybrid micro-parallel computers. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006.

CHAPTER 12

THE JHDL DESIGN AND DEBUG SYSTEM

Brent Nelson, Brad Hutchings
Department of Electrical and Computer Engineering
Brigham Young University

JHDL [1, 8] is a CAD environment developed at Brigham Young University for the design, debug, and runtime control of configurable computing applications based on field-programmable gate array (FPGA) technology. Developed roughly between 1997 and 2003 it was made available under an open-source license (*http://www.jhdl.org*) in approximately 2000. The term *JHDL* can refer to one of two things: (1) the JHDL circuit design language itself, or (2) the JHDL CAD system. The JHDL language is a text-based design language for algorithmic construction of structured circuits that is embedded within the Java programming language. JHDL designs are created as Java programs that access JHDL libraries to generate circuits. Within the JHDL CAD environment, circuits can be simulated, netlisted, and downloaded to the reconfigurable computing platform for execution and testing. Additional CAD tools can be built on top of the JHDL infrastructure to support higher-level circuit construction, optimization, and debugging tasks. One of the most unique features of JHDL is its runtime environment, which provides a unified simulator/hardware debugger that can be used to debug and validate a circuit through either simulation or hardware execution, and which contains many features normally found only in source-level software debuggers.

12.1 JHDL BACKGROUND AND MOTIVATION

Historically, FPGA designers have used CAD tools from three sources to develop their designs. The early tools were derived from application-specific integrated circuit (ASIC) tool flows such as schematic capture, HDL synthesis, and so on. Some were invented as new languages or language dialects specifically for FPGA design [4]. Finally, some designers have used general-purpose programming languages (GPLs) to describe FPGA circuitry [3, 9]. Although there are good reasons behind all three tool approaches, the case for using GPLs for FPGA design is quite compelling. Compared to the other two alternatives (schematic capture and HDL synthesis), GPLs are much more accessible to a larger set of users and can be applied to a much broader set of problems. In addition, GPL programming environments are less expensive, more widely available, and more mature (less buggy) than the other two alternatives.

Within the realm of GPL-based design tools, a range of approaches as well as design abstraction levels can be (and have been) supported. Sea Cucumber [11] is representative of high-level tools that *compile* standard programming language descriptions into hardware. In this case, the tie to GPLs is simply that the input specification syntax used is based on a GPL. A different approach is represented by *structural* design languages that leverage GPL language constructs to assist the user in creating a circuit from a set of building blocks (gates, wires, etc.).

JHDL is an *embedded design language* based on the Java GPL and is a structural design tool. Embedded languages like JHDL are specialized application programming interfaces (APIs) where user-defined classes and function overloading are carefully used to create the illusion of a customized circuit design language within the GPL environment. APIs allow designers to build circuits by declaring interfaces and interconnecting gates and modules, all in a structural way. Embedding does this without making any modifications to existing language syntax, which is an important point because modifying the GPL syntax negates most of the advantages of the embedding approach. Examples of past embedded languages include PAMDC [2] and Spyder [9].

As a structural design tool, JHDL constructs circuits from library primitives with the help of provided Java methods (subroutines) and module generators whose execution produces a circuit graph. This graph is then available for manipulation, including simulation and netlisting. In this era of behavioral synthesis, why are we still interested in structural design? There are three answers to this question. First, when working with FPGAs, structural design techniques often still result in circuits that are substantially smaller and faster than those developed using only behavioral synthesis tools. Second, for many applications found in the reconfigurable computing arena (especially where control over circuit placement is required), structural capture is simply a faster, easier to learn, and more effective way to design an application. Thus, it has a place in any high-performance FPGA design tool kit. Third, the circuit graph produced by the execution of a JHDL description is amenable to a variety of modifications prior to netlisting. For example, it can be programmatically modified to insert debug support features, it can be instrumented to support runtime profiling and monitoring of the final hardware, and it can be modified to support checkpointing (extraction of the hardware computation's state for later restoration) and therefore support context switching of designs on and off a configurable computing platform. None of these features are as readily performed using other approaches, especially behavioral synthesis approaches.

An overview of the design process for JHDL-based design is presented in Figure 12.1. As shown, a collection of JHDL class libraries provides the foundation for all JHDL designs. These libraries contain, at a minimum, Java classes representing primitive circuit elements. Layered on top of the device primitives library are additional libraries that contain subroutines to programmatically generate higher-level circuits from the primitives (known variously as *module generators*). A user creates a JHDL design by writing a Java program that instances these library primitives or calls the module generator subroutines that, in turn, instance primitives.

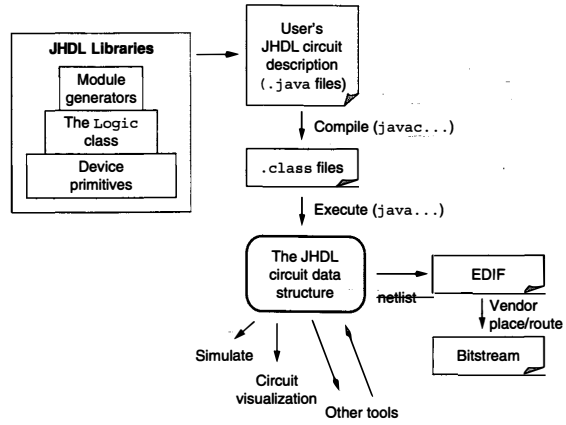


FIGURE 12.1 ■ An overview of the design process and the JHDL system.

Once a JHDL Java program has been written and compiled to a set of class files, it may be executed. The result is an in-memory data structure representing the constructed circuit in the form of a graph, in which nodes represent circuit elements and wires, and arcs represent connections between them. As shown in Figure 12.1, once this data structure has been built, various CAD tools can be applied to it to accomplish simulation, netlisting, or other desired activities. Of interest is that tools can be used to modify the JHDL circuit data structure prior to netlisting or simulation. This is shown in the figure by the arrow leading from "Other tools" up to "The JHDL circuit data structure." These modifications can be for purposes of adding debug or in-circuit monitoring features, and so on.

12.2 THE JHDL DESIGN LANGUAGE

As noted, because JHDL is *embedded*, two mechanisms are used to create the illusion of it as a customized circuit design language: classes and function overloading. The predefined classes provided by JHDL represent primitives, such as gates and wires, so one design method is to simply create instances of these primitives using the Java `new` construct. Beyond this, function overloading provides a higher level of design abstraction by allowing the designer to call parameterized functions that build the desired circuit out of primitive objects. This section describes these levels of JHDL design from a circuit designer's perspective.

12.2.1 Level-1 Design: Primitive Instantiation

The JHDL primitives library shown in Figure 12.1 is simply a package of Java classes where each class corresponds to a circuit primitive (e.g., AND, OR). Given such a library, the lowest level of JHDL design is to instance primitives from it using `new`.

Listing 12.1 shows a simple design built by instantiating primitives. The first two lines import general JHDL libraries needed by all designs. The third import makes the primitives from JHDL's Xilinx Virtex library available for use in the construction of this design.

The `mux` class is next declared by subclassing (extending) the JHDL `Logic` class. The interface ports (the named inputs and outputs for the cell) are declared using the `CellInterface` mechanism where, for example, `in ("sel", 1)` declares an input port named `sel` that is of width 1 and `out ("q", 1)` declares an output port named `q` that is also of width 1.

Listing 12.1 ■ Multiplexer example using primitive instantiation.

```
import byucc.jhdl.base.*;
import byucc.jhdl.Logic.*;
import byucc.jhdl.Xilinx.Virtex.*;

// This cell is a Java class called 'mux'
public class mux extends Logic {

  // Declare the cell's ports
  public static CellInterface[] cell_interface = {
    in("a", 1),
    in("b", 1),
    in("sel", 1),
    out("q", 1),
  };

  // This is the mux's constructor
  public mux(Node parent, Wire aw, Wire bw, Wire selw, Wire qw) {
   super(parent);
    connect("a", aw); connect("b", bw);
    connect("sel", selw); connect("q", qw);

    // The code below this point is the 'body' of the cell and builds
    //   it from primitive wire and gate objects.

    // Declare and construct local wires
    Wire a1 = new Xwire(this, 1, "a1");
    Wire a2 = new Xwire(this, 1, "a2");
    Wire selbar = new Xwire(this, 1, "selbar");

    // Invert signal "sel"
    new inv(this, selw, selbar);
    // Form AND gates
    new and2(this, aw, selbar, a1);
    new and2(this, bw, sel, a2);

    // Form OR gate for final output
    new or2(this, a1, a2, qw);
  }
}
```

The declaration of the constructor for the `mux` class comes next. This is a standard Java constructor method that can be called to construct a new instance of `mux`. The `connect()` calls associate a given wire with a specific port; for example, the wire parameter `aw` is associated with (connected to) port `a`.

The last section of the constructor instantiates the wires and gates needed to implement the multiplexer logic using Java `new...` calls. The objects being created to build the circuit are implemented by Java classes from the `byucc.jhdl.Xilinx.Virtex` package and represent wires and logic gates.

The problem with using primitive instantiation, as just described, is that the resulting design is specific to a particular primitive library (the example above relies on the `byucc.jhdl.Xilinx.Virtex` package). Designing this way limits the portability of the design between technologies, even when it is based on building blocks as simple as individual Boolean gates. Another problem with this design style is that it was specifically written for a multiplexer that has single-bit inputs and outputs—in essence, it is a fixed netlist. The `Logic` class overcomes these limitations.

12.2.2 Level-2 Design: Using the `Logic` Class and Its Provided Methods

The `Logic` class consists of a large collection of subroutines that can be called to create user logic. Listing 12.2 shows the design of the same multiplexer (Listing 12.1) written using methods of the `Logic` class. The difference between this and the previous design is that at the bottom of the constructor, rather than primitive instantiation, this version uses method calls to build the MUX circuit. These methods are available for our use because the `mux` class extends the predefined `Logic` class. In Listing 12.2, the changes from the previous MUX example are underlined, to show how the portion of the code that actually builds the logic has been changed.

Listing 12.2 ■ MUX example written using `Logic` class.

```
import byucc.jhdl.base.*;
import byucc.jhdl.Logic.*;
import byucc.jhdl.Xilinx.Virtex.*;

// This cell is a Java class called 'mux'
public class mux extends Logic {

  // Declare the cell's ports
  public static CellInterface[] cell_interface = {
    in("a", 1),
    in("b", 1),
    in("sel", 1),
    out("q", 1),
  };

  // This is the mux's constructor
  public mux(Node parent, Wire aw, Wire bw, Wire selw, Wire qw) {
    super(parent);
```

```
    connect("a", aw); connect("b", bw);
    connect("sel", selw); connect("q", qw);

    // The code below this point is the 'body' of the cell and builds
    //  it from Logic class subroutine calls.

    or_o(this, and(aw, not(selw)), and(bw, sel), qw);
  }
}
```

Invoking `and(a,b)` calls `byucc.jhdl.Logic.and(a,b)`, which is a subroutine that builds the desired logic (an AND gate) and returns a reference (pointer) to the output wire it created for the gate. This wire can then be used as an input to the `or_o()` call, which creates a 2-input OR gate.[1]

In addition to less verbosity, tremendous power derives from using methods (subroutines) to build circuitry in this manner. OR methods with as many inputs as desired can be created to accommodate any size OR gate and can be written to accommodate input/output wires of any width. Thus, the overloaded `or()` subroutine can handle requests for 2-input OR gates with single-bit inputs/outputs as well as requests for 8-input OR gates with 32-bit inputs/outputs.

The `Logic` class methods accomplish this using JHDL `Techmapper` classes. Figure 12.2 shows that when user code calls a `Logic` method, that method ultimately calls a `Techmapper` class object to do a technology-specific implementation of the logic it has determined should be built, and the `Techmapper` object ultimately maps the resulting logic to technology-specific primitives. This means that designs created using `Logic` class methods are completely technology independent—retargeting a design created using `Logic` to a different technology is as simple as instructing the `Logic` class object to call on a different technology's `Techmapper`. To date, `Techmappers` have been written at Brigham Young University for the Xilinx: 4K, Virtex, Virtex-II, and Virtex-II Pro technologies.

The `Logic` class contains methods to build gates, wires, registers, memories, multiplexers, adders, subtracters, and shifters, as well as methods for manipulating wires: concatenation, slicing, and so forth. Users are encouraged to use the `Logic` style of Listing 12.2 instead of the primitive style of Listing 12.1 whenever possible, as primitive instantiation is typically used only for taking advantage of device-specific features such as clock managers and memories.

[1] Note that some of the function calls have an `an_o` suffix. Functions with this suffix instantiate the gate using the provided input and output wires. Functions without this suffix instantiate both the gate and an output wire that is connected to it. In either case, the output wire is returned by the function. This approach reduces verbosity by eliminating the need to declare and construct intermediate wires.

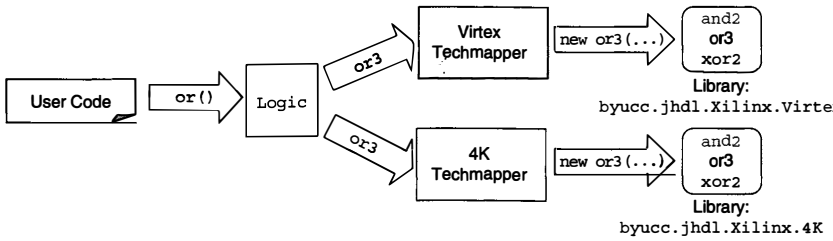


FIGURE 12.2 ■ The relationship of user code, `Logic` class methods, and `Techmapper` objects.

12.2.3 Level-3 Design: Programmatic Circuit Generation (Module Generators)

The creation of programmatic circuit generators (module generators) is a natural extension of the techniques employed by the `Logic` class. That is, Java-based subroutines that intelligently create complex hardware modules based on build time–supplied parameters can be created by any JHDL user. A very simple example that illustrates parameterized design is shown in Listing 12.3.

Listing 12.3 ■ n-bit full adder example.

```
// This design assumes the existence of a FullAdder JHDL design
//  which it instances repeatedly to build an n-bit adder.
import byucc.jhdl.base.*;
import byucc.jhdl.Logic.*;
public class NBitAdder extends Logic {
 public static CellInterface[] cell_interface = {
  param("n", INTEGER),
  in("a", "n"),
  in("b", "n"),
  out("sum", "n+1")
 };

 public NBitAdder(Node parent, Wire a, Wire b, Wire sum) {
  super(parent); // Always call super-constructor
  int width = a.getWidth(); // Get the width of the 'a' wire
  bind("n", width);
  connect("a", a);   connect("b", b);   connect("sum", sum);

  // Create intermediate carry wires as a multi-bit wire
  Wire carries = wire(width);

  // Build and connect together needed full adders
  // The gw() method calls pull individual bits
  //  out of multi-bit wires. The gnd() method returns
  //  a single constant '0' wire.
```

```
    for (int i=0; i < width; i++) {
      if (i==0)
        new FullAdder(this, a.gw(i), b.gw(i), gnd(),
                  sum.gw(i), carries.gw(0));
      else
        new FullAdder(this, a.gw(i), b.gw(i), carries.gw(i-1),
                  sum.gw(i), carries.gw(i));
    }
    buf_o(carries.gw(width-1), sum.gw(width));
  }
}
```

This is an `NBitAdder` design[2] that programmatically constructs a multibit adder using previously designed full adder cells (not shown). In its `CellInterface` declaration, the first line, `param("n", INTEGER)`, declares a parameter `n` that is of type `integer` (similar to a generic in VHDL—see Section 6.1.3). More precisely, `n` is declared to be an instance of the Java class `INTEGER`. All port declarations in the `CellInterface` then use `n` or `n+1` as their width. When `NBitAdder` is constructed, the `bind()` call binds the value of `n` to the width of the `a` wire. Based on this information, `connect()` calls will verify that the wires passed in to the constructor are the correct width for the ports they are being connected to. Finally, the ripple-carry adder body is constructed using a Java `for` loop that creates and interconnects `FullAdder` cells. The final `buf_o()` call connects the top carry-out bit to the most significant bit of the sum.

`NBitAdder` is a trivial example of a module generator, that is, a circuit that can be parameterized according to some set of criteria. Listing 12.4 is a slightly more complex version of the `NBitAdder` design that has been parameterized for pipelining (additions to the original design have been underlined in the source code). Here a single Boolean parameter `"pipe"` is passed into the cell constructor method to control whether a pipeline register is to be placed on the adder output. The main difference between this and the previous design is that the last few lines of the constructor body connect the adder outputs to the cell's outputs through either a register or a buffer, based on the value of the `"pipe"` parameter.

Listing 12.4 ■ n-bit full adder with optional pipelining.

```
... // Same imports as previous NBitAdder design
public class NBitAdder extends Logic {
  public static CellInterface[] cell_interface = {
    ... // Same ports as previous NBitAdder design
  };

  public NBitAdder(Node parent, Wire a, Wire b, Wire sum, Boolean pipe) {
```

[2] The `Logic` class contains a family of multibit adder constructor methods that would normally be called instead of this example design. Nevertheless, this design is presented here to illustrate module generator-like concepts in JHDL.

```
    ... // Main constructor body  same as previous NBitAdder design
    Wire tmpsum;

    // New code is below
    // If desired, insert pipeline latch
    if (pipe)
      (reg_o(tmpsum, sum);
    else
      buf_o(tmpsum, sum);
  }
}
```

On the surface this is similar to what can be accomplished through the use of VHDL generics: the `for-generate` and `if-generate` statements. However, parameterized circuit generation is limited in VHDL, consisting of very simple conditional circuit instantiations that are controlled by a small subset of the language dedicated solely to this purpose. In JHDL, the entire Java language can be brought to bear on this problem and sophisticated algorithms can be used to generate circuits. JHDL module generators exist for counters, comparators, accumulators, arithmetic units (multipliers, dividers, floating-point units, digit serial units), decoders, shift registers, and memories. These have employed, as a part of the module generators' calculations, simple timing and area estimation techniques, recursive tree search computations, file I/O, and the like. Such module generators have been parameterized for features such as number format, rounding/saturation/truncation modes, pipelining granularity, constant encoding methods, and resource usage (serial versus parallel implementation).

12.2.4 JHDL Is a Structural Design Language

Structural design often improves the performance of configurable computing applications because many applications that are FPGA based can benefit from manual placement of at least some parts of the design. Effective manual placement can be achieved only if the overall organization of the circuit is well understood—it is very difficult to manually place circuitry generated by behavioral synthesis.

Placement attributes can be attached to JHDL primitive circuit objects as string properties, to be interpreted by backend tools. To simplify the attaching of these attributes when `Logic` methods are used in circuit building, the `Logic` class also contains a placement API to help in the tasks of (1) mapping gates to lookup tables (LUTs), ALUs, or other atomic FPGA cells, and (2) specifying the relative placement of those cells. For example, to force a collection of gates that implement a 3-input, 1-output logic function into a single LUT, the `map()` call can be used as in:

```
map(a, b, ci, s );
```

This will force the cone of logic with `a`, `b`, and `ci` as inputs and with `s` as output into a single primitive (a LUT for most FPGA technologies). Then that

primitive can be placed by specifying the location of *its output wire* in a `place()` call:

```
place( s, "R0C0.F" );
```

Note that these methods *do not* create logic themselves, but rather pack already created logic into LUTs, which they then physically place. The use of these method calls is technology specific, so a technology-specific `Techmapper` is used to determine their interpretation for the target technology at build time. This placement API acts as a window of opportunity for the user to obtain design assistance from the `Techmapper`. For example, when `map()` is called, the `Techmapper` checks the network of gates for validity (i.e., intermediate fanin or fanout to the network), and, when the circuit is fully constructed, it resolves all placement hints and reports any placement conflicts. In this way placement errors can be detected at the *front* of the tool chain rather than during place and route, which helps minimize design cycles.[3]

12.2.5 JHDL Is a Programmatic Circuit Design Language

That JHDL is also a programmatic circuit design language is perhaps the most powerful and unique feature of GPL-based circuit generation techniques. The key point is that a JHDL description, once compiled, *is* an executable Java program; it is the execution of that Java program that constructs the circuit. This gives JHDL significant advantages over HDL descriptions, which must be *parsed* by a synthesizer and a corresponding circuit then constructed.

With JHDL there is no separation between the code that represents the circuit itself and any code that might be executed to help determine how best to generate it—all of the code in a JHDL description is executable Java. In a language like JHDL there is a very clear separation between circuit generation and computation: Module instantiation is circuit generation and everything else is computation. In contrast, all code written in a VHDL or Verilog design (excepting simulation testbenches) *is* the circuit description—there is no provision for code that can be executed apart from it. This presents difficulties when computations are required, prior to circuit construction, to determine how best to generate the circuit.

At one time, designers often resorted to macro preprocessors with Verilog code to provide `for-generate`- and `if-generate`-like functionality for their designs. Similarly, some designers (including our own students) have often written C or C++ circuit generators that generate VHDL or Verilog code as output in order to work around the lack of an effective compile time computational capability in conventional HDLs. In contrast, JHDL and other GPL-based embedded languages avoid such workarounds because they provide a clean mechanism

[3] In contrast, VHDL annotation approaches for placement are nonstandard and differ from tool to tool (see Section 6.2.1). Also, VHDL placement directives are passed through to the backend tools without performing any error checking such as described previously.

for freely intermixing computational code with circuit descriptions—all based on the general-purpose computational power of the underlying GPL. One could say that languages like JHDL don't need a formal elaboration step as VHDL and Verilog do. Or one could say that the *entire* circuit construction process in JHDL is an elaboration step, albeit a much more powerful one than that provided by HDLs.

Finally, there is no *synthesizable subset* of JHDL, and thus there is no possibility for a mismatch between simulation and synthesis results due to differing CAD tools' interpretation of the description. The same circuit is constructed each time the JHDL code is executed regardless of whether it is intended for simulation or for netlisting.

12.3 THE JHDL CAD SYSTEM

As Figure 12.1 showed, the execution of a compiled JHDL design creates an in-memory structural representation of the JHDL circuit. This is a classical circuit graph where Java objects represent the cells and wires in the circuit and pointers between these objects represent connections and hierarchical parent–child relationships. The figure also showed that this circuit data structure is the entry point for the JHDL CAD system, meaning that all CAD functions and tasks, such as simulation and netlisting, use the circuit data structure via an API provided for this purpose. The result is that it is straightforward to write Java-based CAD tools for interacting with and manipulating the circuit data structure (and therefore the circuit).

12.3.1 Testbenches in JHDL

Because a JHDL design is a Java program, it needs a `main()` routine. It is the program's `main()` routine that usually acts as a testbench for JHDL designs. Listing 12.5 shows such a `main()` testbench that, like most JHDL testbenches, does three things:

- First, it creates an `HWSystem` object that is the top-level container object for the circuit and contains the simulator and netlister objects. The entire user design (testbench and device under test) exists as a child node of `HWSystem` in the resulting JHDL object hierarchy.

Listing 12.5 ■ A sample JHDL testbench.

```
import ...;    // Import needed packages

// Declare testbench class
public class tb_myCell extends Logic implements TestBench {

  static HWSystem hw;   // Declare a HWSystem
  private int aVal, bVal, cinVal;  // Declare some private variables
```

```
  // The main() routine for this Java program
  public static void main(String argv[]) {
    // Step 1: build a HWSystem
    hw = new HWSystem();  // Build a HWSystem
    // Step 2: Build an instance of this testbench
    tb_myCell tb = new tb_myCell(hw, ...); // Pass in some params
    // Step 3: Do something with the circuit now that the testbench
    // and DUT are built.  We can do any one of:
    // 1. Start a simulation
    // 2. Netlist the circuit
    // 3. Traverse or modify the circuit data structure
    // 4. Start a GUI-based CAD system
    // We will do the last - create a GUI-based CAD system
    // Create a new instance of cvt (the Circuit Visualization Tool)
    new cvt( tb );
  }

  // The constructor for this testbench
  public tb_myCell (Node parent, ...);  // Not all params shown
    super(parent);

    // Step 1: Specify (create) a TechMapper for Virtex
    setDefaultTechMapper(new VirtexTechMapper(true));

    // Step 2: Build wires to connect to DUT
    an = wire(1,"an");    bn = wire(1,"bn");    cinn = wire(1, "cinn");
    sn = wire(1,"sn");    coutn = wire(1,"coutn");

    // Step 3: Build mycell (the DUT)
    myCell dut = new myCell(this, an, bn, cinn, sn, coutn, "myCell");
  }
}
```

- Second, as shown in Listing 12.5, it creates a testbench object (which in turn creates the device under test).
- Third, once the JHDL circuit data structure has been created, the `main()` routine can do one of a number of things: (1) start a batch simulation, (2) call on the netlister to netlist the design, or (3) create a GUI-based interface to enable the user to interactively work with the circuit. In Listing 12.5, the `main()` routine starts up `cvt`, a graphical environment for viewing the circuit, simulation, and netlisting.

12.3.2 The `cvt` Class

Class `cvt` is a GUI-based system with widgets for navigating the design hierarchy, starting a simulation of the circuit, generating a netlist of the circuit, and so forth. The actual simulation and netlisting classes are accessed via the `HWSystem` class, and `cvt` makes calls into it to satisfy user requests. The `cvt` class implements a standard event-driven GUI system based on Swing, distributed as a part of the JHDL language. Swing was chosen for its portability and availability on all

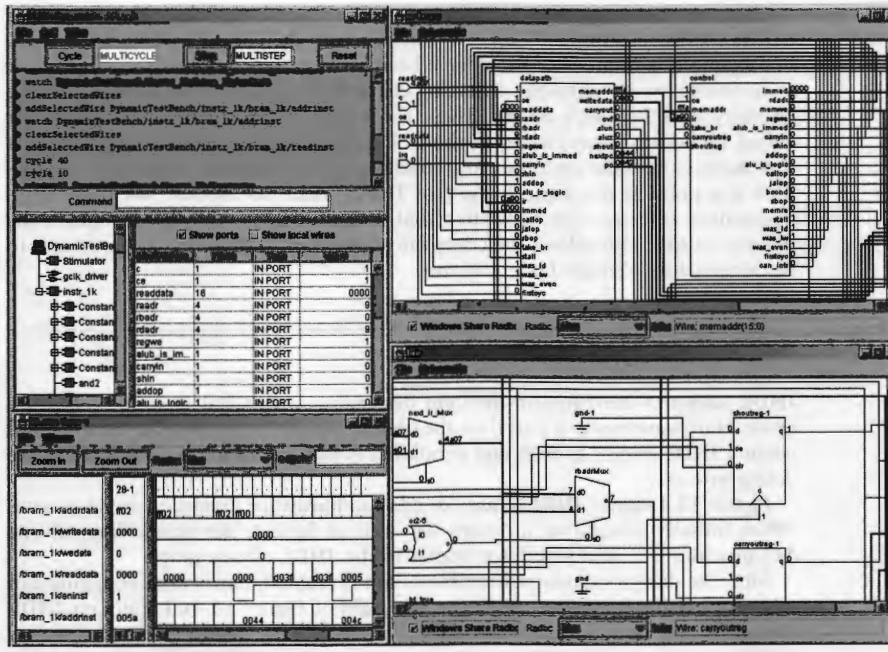


FIGURE 12.3 ■ The JHDL `cvt` GUI.

platforms. Class `cvt` uses the built-in Swing event mechanism for its own internal communication.

Figure 12.3 shows a screenshot of the `cvt` GUI. In the upper left is a text console window where commands may be typed. Menus and buttons above largely duplicate what can be entered in the window. Below the console is a hierarchy navigation tool. On the left is a hierarchy browser; on the right is a list of ports for the currently selected cell along with their current values (if a simulation is in progress). Beneath the browser is a waveform viewer; and on the lower right half of the figure are two different schematic viewers. This screenshot shows that the various parts of the GUI are all contained in a single pane but each can be broken out into an individually sized window if desired.

Unlike with most CAD tools, there is no *standard* JHDL CAD system. Rather, the circuit data structure API provides a mechanism for any program a user might write to interact with the circuit. The `cvt` class is simply one such example. Examples of other programs include stand-alone simulators and netlisters.

Many of the debugging experiments described later in this chapter were carried out by writing custom CAD tools to interact with the circuit data structure API. For example, a number of tools have been written that modify the circuit prior to netlisting by, for example, inserting clock managers or other special-purpose circuitry into the user's design. These tools have also instrumented designs for debug by adding scan chains to them. Finally, complete software applications that interact with the circuit during simulation and execution have been created. This last point is a unique feature of JHDL—once the circuit has been built, application software can be written that communicates with the design via the `HWSystem` API. This allows the complete application (software and hardware) to be deployed as a single Java program.

12.4 JHDL'S HARDWARE MODE

JHDL supports hardware-in-the-loop debugging with what is called *hardware mode*. Hardware mode is based on the observation that much of the data created when a JHDL circuit is built and simulated is also useful in the actual hardware debug process.

Figure 12.4 shows JHDL's dual simulation/hardware execution environment. When initially simulating a design (*left* side of figure), the simulation/runtime API provides `cvt` and simulator access to the JHDL circuit graph.

After the design's configuration bitstream is created, hardware debugging can take place using hardware mode (on the right of Figure 12.4). Loading a JHDL design in `cvt` now performs two steps: (1) the JHDL design is constructed as usual to create the internal circuit representation, and (2) the bitstream is configured onto the specified FPGA platform. Using the same `cvt` GUI as before, the user can advance execution of the design via the simulator control buttons or via commands on the command line. However, instead of cycling the simulator, these actions cause `cvt` to send clocking commands to the FPGA platform through the board's driver. After a clock command is executed, the state of the FPGA platform is retrieved using readback and *back-annotated into the JHDL circuit data structure*.

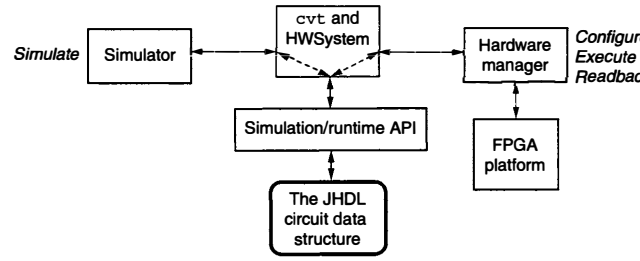


FIGURE 12.4 ■ The JHDL unified simulation/hardware execution environment.

The simulator is then used to compute the steady state of all combinational nodes in the circuit as a function of these state values, and a complete picture of the hardware execution state is now present in the JHDL circuit data structure. As a result, `cvt` can query and display the state of the circuit as normal, just as it would if the circuit were being simulated. In this case, however, it displays hardware signal values rather than simulated signal values.

JHDL's hardware mode is readily adapted to new hardware platforms given programmatic methods that exist for communicating with the board. The following capabilities are required for adaptation:

- *Configuration:* This is needed to configure the FPGA(s) on the hardware platform with bitstreams.
- *Clock control:* One or more subroutines are required to single- or multistep the clock on the board.
- *Readback:* This is necessary to read back the state from the FPGA(s) on the board.[4]

Given these capabilities, JHDL can easily be extended to communicate with the board for hardware mode operation.[5] This is achieved by modifying a thin layer of Java translation code so that standard JHDL methods can communicate with the specific C-based device driver subroutines for the board.

12.5 ADVANCED JHDL CAPABILITIES

A variety of design and debug tools have been built on top of JHDL. A few of these are described in the following sections.

12.5.1 Dynamic Testbenches

Some of the power of JHDL derives from its extensive use of the Java feature called *reflection*. The Java reflection API provides a set of methods that a Java program can use to examine the structure of a Java `.class` file. By reflecting on a Java class, a program can determine the names and type signatures of all methods in it and, if desired, dynamically load the class file and construct an object of the class.

[4] To date, JHDL supports hardware mode only on Xilinx platforms, because they contain a readback capability. Experiments have also been done to determine the cost of adding a scan chain to user designs for this purpose when readback is not available [12].

[5] An additional capability would also prove very useful—loading state into the FPGA. In the case of Xilinx FPGAs (the focus of the JHDL hardware mode work), this can be done by modifying the configuration bitstream appropriately and then reconfiguring the FPGA with that bitstream. Thus, this capability is not listed as a strict requirement in the list. A number of the debug experiments described in the next section performed bitstream modification to load state into an FPGA, but would have benefited from a simpler mechanism that did not require a reconfiguration of the FPGA.

The JHDL `dtb` class is a general-purpose testbench tool that uses reflection to automatically perform testbench functions, eliminating the need, in most cases, for the user to write code for constructing the testbench. The user runs `dtb` and specifies the name of the circuit to be constructed on the command line. `dtb` examines the corresponding file (`FullAdder.class`, for example) using reflection to determine the parameters required by its constructor. It then creates the necessary wires and calls the constructor to build an instance of the specified class, connecting it to the wires it created. When `dtb` is used, the `dtb` object itself performs all of the services required of a testbench. For example, it examines the constructed circuit, determines the clocking required, and sets up the clock for simulation. In addition, when everything has been constructed, it brings up `cvt` so that design simulation and netlisting can proceed as usual. All that is required of the user is to provide the simulation stimulus either interactively or via a script.

12.5.2 Behavioral Synthesis

Sea Cucumber [11] is a behavioral synthesis tool that was built on top of the JHDL framework and accepts a behavioral description written in Java that is compiled into bytecodes by any standard Java compiler. It parses these byte codes, discovers instruction-level parallelism, performs other common optimizations, and then synthesizes a circuit by invoking calls to the JHDL Logic library. Advantages provided by the JHDL framework include access to JHDL visualization and debugging tools to verify Sea Cucumber designs and access to JHDL netlisting modules so that the synthesized JHDL circuitry can be converted into netlists for place and route by vendor software. In fact, all of the previously mentioned JHDL features are available to Sea Cucumber, including hardware mode, dynamic test benches, and the like.

A behavioral debugger, also developed in conjunction with Sea Cucumber [7], allows the user to debug fully optimized code in the context of the original user description. It does this by traversing the JHDL circuit structure to retrieve circuit values and presenting them to the user in the context of the JHDL CAD framework.

12.5.3 Advanced Debugging Capabilities

Much of the power of JHDL comes by exploiting a single FPGA feature (readback) to access internal FPGA state and present the data to the user in some form. Because of this enhanced visibility, the current JHDL debugging environment has proved to be effective for verifying and debugging large, complex applications. However, much more powerful debugging capabilities can be achieved if a small part of the FPGA is reconfigured to implement supplemental circuitry to aid debug and validation. This is similar in spirit to the `"-g"` flag used in conventional software compilation where the compiler can enable debugging by inserting additional code. Because FPGA hardware is reconfigurable, any inserted debugging circuitry can be removed when the application is ready for deployment.

Some of the advanced debugging features that are possible via embedded debug circuitry include:

- Signals can be automatically routed to external I/O pins for viewing.
- Unused FPGA circuitry and memory can be used to implement "probe" circuits that sample and store circuit activity during circuit execution.
- Unused FPGA hardware can be used to implement complex, real-time hardware breakpoints.

As long as designers must manually modify their designs in order to embed debug circuitry, these powerful techniques may go unused. The best way to overcome this is to automate the process of synthesizing and embedding debug circuitry into user circuitry—a task best performed directly by the CAD tool environment. The ability to use a debugging tool as an integrated part of the design environment, tied to the original design specification and accessed using standard user interfaces, makes this a powerful and convenient way to develop and verify a design.

As a part of DARPA-funded research at Brigham Young University, researchers investigated a variety of advanced debug mechanisms using JHDL, all of them were aimed at providing a debug system with capabilities similar to those found in software development systems and that are significantly easier to use than manual methods. A few of these mechanisms are described in the following subsections.

Debug circuitry synthesis

In software debugging using the *gdb* symbolic debugger or similar tools, it is not uncommon for the user to temporarily change variable values as a way of determining how the program would behave *if* the variable had that different value. The work described by Graham [6] demonstrated a similar capability for hardware. First, JHDL was used to perform a readback of the FPGA's state. Changes were then made to the bitstream to reflect the user's choice for the new circuit state, and the bitstream was configured back into the FPGA. Upon resumption, the system was seen to continue execution from the previous point but with changed state values.

As another example, in the work described by Graham et al. [5], JHDL and JBits were used together to modify FPGA design bitstreams on the fly in order to rewire embedded logic analyzers to user logic in a placed-and-routed design—all within a few seconds of a mouse click. The collected data could then be viewed in the original design environment using the built-in JHDL GUI framework.

When these features first appeared in JHDL, there were no equivalent commercial debugging tools available. However, with the passage of time, commercial offerings have improved, incorporating some of the features of the original JHDL system. Altera's SignalTap and Xilinx's ChipScope now provide convenient ways to integrate customized logic analyzers into user designs (these are implemented with unused programmable circuitry), and offer separate tools that emulate a logic analyzer display on the PC's monitor. However, JHDL still differs from these products in the level of integration it provides (the debug

environment *is* the design environment). Perhaps Synplicity's Identify tool comes closest to JHDL in this regard because it provides the ability to view some circuit behavior in the original VHDL context. Still, none of the commercial offerings allow the user to simultaneously integrate logic analyzers, display these results in the original design environment, and modify the current state of the circuit during debug.

Checkpointing, context switching, and remote access

Checkpointing is defined as saving the state of a computation in a way that the computation can later be restarted from that same point. It is often used in software to allow a long-running computation such as a simulation to be restarted from a known point if, for example, the system it is running on goes down. The concept of readback can easily be extended to extract the state of the entire FPGA platform.

Once this is done, checkpointing of FPGA-based computations can be supported. To do this, the JHDL `HWSystem` was augmented not only to retrieve the state of the FPGA but also to retrieve the state of all memory elements on the hardware platform (FIFOs and memories) and save that information to disk. Later, the state could be retrieved from disk and loaded back onto the FPGA platform, whereupon execution would continue from the time of the checkpoint. What is important is that a *simulation* checkpoint could be loaded onto the FPGA platform and *hardware execution* could be continued from that point. Likewise, a *hardware execution* checkpoint could be loaded into JHDL and a *simulation* continued from that point. With the availability of checkpointing in JHDL, it then became possible to time-share an FPGA platform using context switching (swapping an application off the platform to make room for another). Experiments conducted at Brigham Young University on checkpointing and context switching are described by Landaker et al. [10], to which the interested reader is referred for more information and results.

Finally, JHDL was also modified to permit remote access to an FPGA platform. In this work, the `cvt` and `HWSystem` classes were extended to include a client–server capability so that hardware mode communications with an FPGA platform could be conducted over a network.

12.6 SUMMARY

JHDL is currently in use in a variety of research projects, from module generators systems to behavioral synthesis systems to microarchitectural simulation systems. By providing a framework for the construction, simulation, netlisting, and hardware debug of FPGA-based designs, JHDL allows researchers to focus on tasks other than recreating the infrastructure that JHDL provides. Of particular importance in this regard, is that JHDL provides a target for use by synthesis tools with its primitive libraries and its `Logic` and `Techmapper` classes.

JHDL has been in use since approximately 1998 and was released under an open-source license (*http://www.jhdl.org*) in approximately 2000. Potential