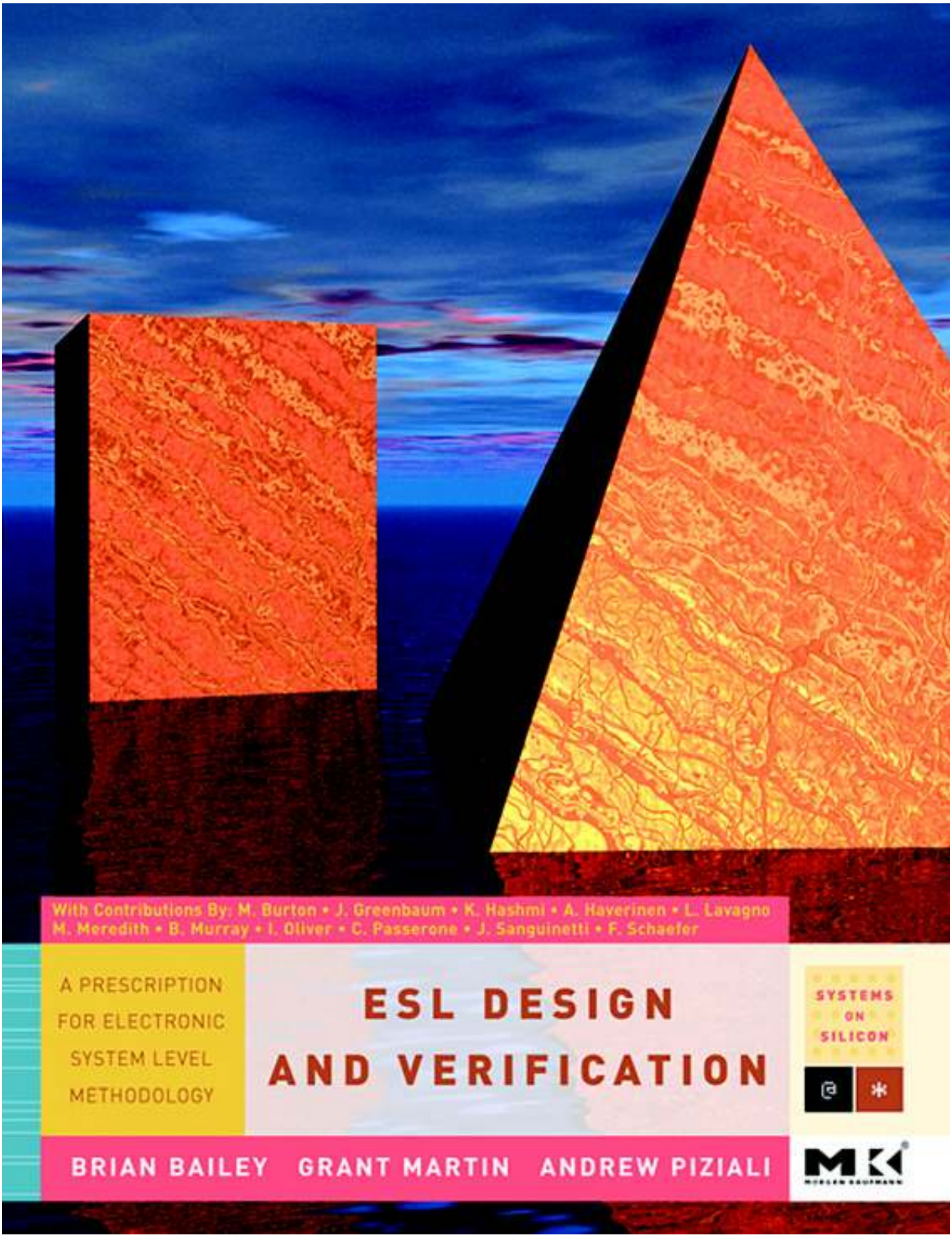


With Contributions By: M. Burton • J. Greenbaum • K. Hashmi • A. Haverinen • L. Lavagno
M. Meredith • B. Murray • I. Oliver • C. Passerone • J. Sanguinetti • F. Schaefer

A PRESCRIPTION FOR ELECTRONIC SYSTEM LEVEL METHODOLOGY

ESL DESIGN AND VERIFICATION

SYSTEMS ON SILICON

BRIAN BAILEY GRANT MARTIN ANDREW PIZIALI

MK MORGAN KAUFMANN

ESL Design and Verification

A Prescription for Electronic System-Level Methodology

The Morgan Kaufmann Series in Systems on Silicon
Series Editor: Wayne Wolf, Princeton University

The rapid growth of silicon technology and the demands of applications are increasingly forcing electronics designers to take a systems-oriented approach to design, This has led to new challenges in design methodology, design automation, manufacture and test. The main challenges are to enhance designer productivity and to achieve correctness on the first pass. *The Morgan Kaufmann Series in Systems on Silicon* presents high-quality, peer-reviewed books authored by leading experts in the field who are uniquely qualified to address these issues.

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

Readings in Hardware/Software Co-Design
Edited by Giovanni De Micheli, Rolf Ernst, and Wayne Wolf

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

Comprehensive Functional Verification
Bruce Wile, John Goss, and Wolfgang Roesner

Customizable Embedded Processors: Design Technologies and Applications
Edited by Paolo Ienne and Rainer Leupers

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

Designing SOCs with Configured Cores: Unleashing the Tensilica Diamond Cores
Steve Leibson

VLSI Test Principles and Architectures: Design for Testability
Edited by Laung-Terng Wang, Cheng-Wen Wu, and Xiaoqing Wen

Contact Information

Charles B. Glaser

Senior Acquisitions Editor
Elsevier
(Morgan Kaufmann; Academic Press; Newnes)
(781) 313-4732
c.glaser@elsevier.com
http://www.books.elsevier.com

Wayne Wolf
Professor
Electrical Engineering, Princeton University
(609) 258-1424
wolf@princeton.edu
http://www.ee.princeton.edu/~wolf/

ESL Design and Verification

A Prescription for Electronic System-Level Methodology

Brian Bailey
Grant Martin
Andrew Piziali

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

ELSEVIER

Morgan Kaufmann Publishers is an imprint of Elsevier

MORGAN KAUFMANN PUBLISHERS

Senior Acquisitions Editor	Charles B. Glaser
Publishing Services Manager	George Morrison
Senior Project Manager	Brandy Lilly
Assistant Editor	Michele Cronin
Cover Design	Eric DeCicco
Composition	Cepha Imaging Pvt. Ltd.
Copyeditor	Graphic World
Proofreader	Graphic World
Indexer	Graphic World
Interior Printer	Maple-Vail Book Manufacturing Group
Cover Printer	Phoenix Color

Morgan Kaufmann Publishers is an imprint of Elsevier.
500 Sansome Street, Suite 400, San Francisco, CA 94111

This book is printed on acid-free paper.

Library of Congress Cataloging-in-Publication Data
Bailey, Brian
ESL design and verification: a prescription for electronic system-level methodology/by Brian Bailey, Grant Martin, Andrew Piziali — 1st ed.
p. cm — (Morgan Kaufmann series in systems on silicon)
includes bibliographical references and index
ISBN-13: 978-0-12-373551-5 (hc)
ISBN-10: 0-12-373551-3 (hc)
1. Systems on a chip—Design and construction. I. Martin, Grant. II. Piziali, Andrew. III. Title. IV. Title: Electronic system-level design.
TK7895.E42M365 2007
621.3815—dc22
2006103541

ISBN 13: 978-0-12-373551-5
ISBN 10: 0-12-373551-3

For information on all Morgan Kaufmann publications,
visit our Web site at www.mkp.com or www.books.elsevier.com

Printed in the United States of America
07 08 09 10 11 5 4 3 2 1

CONTENTS

3 Evolution of ESL Development 35

This page intentionally left blank

FOREWORD: ESL FROM THE TRENCHES

Over the past 15 years, electronic system-level (ESL) design has come to be the most attractive and fascinating concept in the chip design industry. Adoption of the ESL design methodology for complex systems-on-chips (SoC) appears around nearly every corner in the labyrinth of evolving design technologies, and meets the demanding technological challenges arising in the competitive field of SoC design.

As SoCs are getting more complex and design process technology rapidly approaches 22 nanometers, product development time is lengthening while the demands of time-to-market (TTM) are shortening. Because the contradictions between ever-increasing SoC complexity, reduction of TTM, and maintaining low development costs are deepening over the entire SoC industry, flawless and epoch-defining design methodologies are now more necessary than ever. ESL design methodology is seen as a promised savior, but has yet to come to life.

To apply an ESL design methodology, the correct tools and highly skilled human resources are deemed necessary, both of which require a substantial investment from a design company. Also, complications arise when digital convergence accelerates while products diversify; thus, co-development of hardware and software becomes crucial. To build up their core design competency, many SoC companies emphasize embedded software development as a priority. However, as yet there have been no significant advances in embedded software development achieved by applying ESL design technology to the complex SoC.

It is not unusual to hear disheartening tales from the engineering trenches regarding ESL, despite much optimism. The biggest obstacle to deploying ESL technology is the lack of a measurable key performance index to justify the company's investment in an evolving methodology. It is necessary to discover a means of reducing the modeling effort drastically because a large effort to create transaction-level models would cancel the benefits of ESL. Amazingly, there are not many ESL tools available to develop the embedded software for SoCs because of inadequate simulation speed. Tool interoperability is also necessary to maximize the return-on-investment for the adoption of ESL. Other important considerations are the lack of back-annotated design constraints and validation of the correlation between behavioral and register transfer-level models. And the stories from the trenches continue indefinitely.

Finally, the biggest obstacle facing ESL is an inappropriate bias—ESL is redundant, inaccurate, unclear, difficult to understand, and so forth—that stems from the vague

definition of ESL. Hence, the SoC designer adopts ESL in their design flow only reluctantly.

This book, written by Brian Bailey, Grant Martin, and Andrew Piziali, explains the various definitions and taxonomy of ESL in a comprehensive way, thus providing a solid baseline for understanding ESL for either the novice or the experienced engineer. It also describes the various ESL tools used in the past as well as the present, giving readers a general overview of ESL evolution. Special attention is paid to embedded software development from the ESL design methodology perspective. Students, researchers, hardware and software design engineers, and engineering managers will find this book to be a valuable asset for insightful discussions.

With this book, the position of ESL design methodology is well consolidated, and the SoC industries will obtain real benefits from using innovative ESL technologies with supporting tools. I applaud this valuable contribution.

Soo Kwan Eo
Senior Vice President
SoC R&D Center, System LSI Division
Semiconductor Business, Samsung Electronics Co, LTD
Korea
August, 2006

AUTHORS' ACKNOWLEDGMENTS

Brian Bailey thanks his parents for the sacrifices they made and for giving him such a good start to life; and those close to him who have given him the courage, strength, and energy necessary to complete this book.

Grant Martin thanks his wife Margaret Steele and daughters Jennifer and Fiona. He would also like to acknowledge all his colleagues over the years at Burroughs, BNR/Nortel, Cadence Design Systems, and Tensilica, Inc., without whom his knowledge of system-level design and ESL would be nonexistent.

Andrew Piziali thanks his wife Debbie and son Vincent for the time needed to contribute to this work. His mentors Tom Kenville and Vern Johnson introduced him to "diagnostics development," spurring his long-term interest in design verification. Finally, he thanks his co-authors and contributing authors for the opportunity to learn from their diverse experience.

All three authors would like to thank their several contributors: Mark Burton, Jack Greenbaum, Kamal Hashmi, Anssi Haverinen, Luciano Lavagno, Mike Meredith, Bill Murray, Ian Oliver, Claudio Passerone, John Sanguinetti, and Florian Schäfer, without whom this book would not have been possible. We (the authors and contributors) would also like to thank Clive "Max" Maxfield, Sumit Gupta, and Alberto Sangiovanni-Vincentelli for diagrams in Chapter 3, and Guy Bois and Peter Ashenden for their contributions about education and ESL in Chapter 14. We would also like to thank David Black and Frank Schirrmeister for help with Chapter 3. We owe special thanks to Soo Kwan Eo for writing the foreword to the book.

This book had some early readers who made useful and important suggestions for improvement and clarification. The authors and contributors would like to thank Harry Foster, Doug Matzke, and Gary Smith for this contribution. In addition, several people at Elsevier, Morgan Kaufmann, and Graphic World Publishing Services were of special help and support in the book writing and production process, from the initial idea through to the final product: Chuck Glaser, Denise Penrose, Michele Cronin, Brandy Lilly, and Paul Sobel. We would also like to thank everyone else at the publishers who helped make this book a reality.

Finally, in order to prepare for this book, the authors conducted a survey and questionnaire on ESL in the summer of 2005, and benefited greatly from the responses to that survey in developing the outline and plans for the book. We would like to thank those who responded to the survey: Anssi Haverinen, Clive Maxfield, Mark Burton, John Sanguinetti, Wolfgang Nebel, Wolfgang Ecker, Kamal Hashmi,

Miron Abramovici, Scott Sandler, Mark Lippett, Jeff Jussel, Richard Yeh, Neal Stollon, Luciano Lavagno, Dave Harris, Trevor Wieman, Mitch Dale, Bill Murray, Gary Smith, and Daya Nadamuni. One might notice that several of the contributors responded to the survey, and then agreed to make a contribution to the book. Their survey responses were so insightful that we felt strongly that they could make a valuable contribution here, and you will see their input throughout the rest of the book.

About the Authors

Brian Bailey is an independent consultant helping EDA and system design companies with technical, marketing and managerial issues related to verification and ESL. Before that he was with Mentor Graphics for 12 years, with his final position being the Chief Technologist for verification, Synopsys, Zycad, Ridge Computers, and GenRad. He graduated from Brunel University in England with a first-class honors degree in electrical and electronic engineering.

Brian is the co-editor of the book *Taxonomies for the Development and Verification of Digital Systems* (Springer, 2005) and the executive editor and author for *The Functional Verification of Electronic Systems: An Overview from Various Points of View* (IEC Press, 2005). He also authored a chapter of the book *System-on-Chip Methodologies & Design Languages* (Kluwer 2001). He has published many technical papers, given keynote speeches at conferences, performed seminars around the world, and been both a contributor and moderator of panels at all of the major conferences.

Brian established the functional verification track in the DesignCon conferences, which has quickly grown to be one of the major tracks of the conference. He also serves on the technical program committees of many major conferences, including the Design Automation Conference (DAC). He chairs the interfaces standards group within Accellera and has in the past chaired other standards groups in Accellera and VSIA.

Brian is primarily interested in the specification, simulation, and analysis of embedded systems and today is moving into the problems associated with, and solutions necessary for, multiprocessor systems.

Grant Martin is a Chief Scientist at Tensilica, Inc. in Santa Clara, California. Before that, Grant worked for Burroughs in Scotland for 6 years; Nortel/BNR in Canada for 10 years; and Cadence Design Systems for 9 years, eventually becoming a Cadence Fellow in their Labs. He received his Bachelor's and Master's degrees in Mathematics (Combinatorics and Optimisation) from the University of Waterloo, Canada, in 1977 and 1978.

Grant is a co-author of *Surviving the SOC Revolution: A Guide to Platform-Based Design*, and *System Design with SystemC*, and a co-editor of the books *Winning the SoC Revolution: Experiences in Real Design*, and *UML for Real: Design of Embedded Real-Time Systems*. In 2004, he co-wrote with Vladimir Nemudrov the first book on SoC design published in Russian by Technosphera, Moscow. Recently he co-edited

Taxonomies for the Development and Verification of Digital Systems (Springer, 2005), *UML for SoC Design* (Springer, 2005), and the two-volume *Electronic Design Automation for Integrated Circuits Handbook* (Taylor and Francis/CRC Press, 2006). He has also written or co-written chapters in several other books.

He has also presented many papers, talks, and tutorials, and participated in panels, at a number of major conferences. He co-chaired the VSIA Embedded Systems study group in the summer of 2001, and was co-chair of the Design Automation Conference (DAC) Technical Programme Committee for Methods for 2005 and 2006.

His particular areas of interest include system-level design, IP-based design of SoC, platform-based design, and embedded software. Grant is a Senior Member of the IEEE.

Andrew Piziali is an industry veteran design verification engineer with 24 years' experience verifying mainframes, supercomputers, and microprocessors with StorageTek, Inc., Amdahl Corp., Evans and Sutherland, Convex Computer Corp., Cyrix Corp., Texas Instruments, Inc., and Transmeta Corp., and developing verification methodologies, technologies, and products with Verisity Design, Inc. and Cadence Design Systems. Having an avid interest in coverage-driven verification, in 2004 he authored the book *Functional Verification Coverage Measurement and Analysis*. He has authored a number of papers and is an active contributor to the Design Automation Conference (DAC), DesignCon, DVCon, and the IBM Haifa Verification Conference. Andrew is currently employed by Cadence Design Systems as a Verification Application Specialist, focusing on verification planning and management.

About the Contributors

Mark Burton received his B. Eng. from the University of Warwick, and his Ph.D. in AI and Education from Leeds University. He was an engineering manager at ARM, and also chair of OSCI's transaction-level modeling (TLM) working group. Recently he founded GreenSoCs, as both a consultancy and an open-source community centered on SystemC, and is also the chair of OCP-IP's system-level design working group.

Jack Greenbaum is the Director of Embedded Software Development in the Advanced Products Group at Green Hills Software, where he is involved in the development of complex embedded system and RTOS infrastructure. His career has spanned both EDA and software development tools at leading semiconductor and embedded software companies. His research interests include ESL tools and environments, performance analysis for embedded systems, and reconfigurable computing. Jack earned a BS in Computer Science and MS in Electrical and Computer Engineering at the University of California, Santa Barbara.

Kamal Hashmi is a co-founder and VP of Research and Development at SpiraTech, Ltd., in Manchester, U.K. Kamal is an expert in ESL design tools and languages, and interface-based design methodologies. He has been a major contributor to the VSI System Level Design working group and written a number of papers on system-level design. He has previously worked in data management, simulation, and test before moving to system design languages and methodologies at ICL/Fujitsu. Kamal is a Chartered Mathematician and holds an Honours degree in Mathematics from Leeds University.

Anssi Haverinen works with Texas Instruments as a system architect in 3G wireless technology in San Diego, California. Previously, he worked with Nokia from 1992 to 2006 in several roles, the latest as a system design manager for US-CDMA cell phone platforms. He has actively driven the development of ESL methodology and open standards, especially in transaction-level modeling, in his companies and in the standards forums of OCP-IP, OSCI, and VSIA. Anssi holds an M.Sc.(EE) from Tampere University of Technology, Finland, in microelectronics.

Luciano Lavagno received his Ph.D. in EECS from U.C. Berkeley in 1992 and from Politecnico di Torino in 1993. He is a co-author of two books on asynchronous circuit design, a book on hardware/software co-design of embedded systems, and of over 160 scientific papers, and serves on the technical committees of several international conferences in his field. Between 1993 and 2000 he was the architect of

the POLIS project, which developed a complete hardware/software co-design environment for control-dominated embedded systems. He is currently an Associate Professor with Politecnico di Torino, Italy and a research scientist with Cadence Berkeley Laboratories. His research interests include the synthesis of asynchronous and low-power circuits, the concurrent design of mixed hardware and software embedded systems, and compilation tools and architectural design of dynamically reconfigurable processors.

Mike Meredith is the Vice President of Technical Marketing for Forte Design Systems. He also serves as the president of the Open SystemC Initiative (OSCI). He has over 10 years' embedded systems experience in the biomedical and industrial automation industries. He began working in the EDA industry more than 15 years ago, creating printed circuit board layout and schematic capture tools, was a founder of Chronology Corporation, and one of the authors of the TimingDesigner timing diagram entry and analysis tool. He is the holder of three U.S. patents in the areas of timing diagrams and timing analysis of electronic circuits. He is currently engaged in the development of SystemC and behavioral synthesis tools using SystemC.

Bill Murray is a technical and business consultant to EDA, semiconductor, and systems companies. He received his M.Sc. in Applied Solid State Physics from Brighton Polytechnic, U.K in 1972, and his B.Sc. (Hons) in Applied Physics from Sussex University, U.K. in 1971. He has held engineering and technical marketing positions in Texas Instruments, VLSI Technology, and Cadence Design Systems, in Germany, the United Kingdom, and the United States. He has been involved in ESL design methodologies since 1996, when he joined the Alta Group of Cadence.

Ian Oliver received his doctorate from the University of Kent at Canterbury, U.K., in 2001. He has been working with Nokia Research for the past 7 years on the use of UML and formal methods for the specification of real-time and embedded systems.

Claudio Passerone received the M.S. degree in Electrical Engineering from Politecnico di Torino, Italy and the Ph.D. degree in Electrical Engineering and Communication from the same university, in 1994 and 1998, respectively. He is currently a researcher in the Electronics Department of Politecnico di Torino. His research interests include system-level design of embedded systems, electronic system simulation and synthesis, and reconfigurable computing. Dr. Passerone is a co-author of a book on hardware/software co-design of embedded systems, has published over 50 journal and conference papers, and served on technical committees of DATE and ISCAS.

John Sanguinetti received his Ph.D. in Computer and Communication Sciences from the University of Michigan in 1977. Since that time he has been active in computer architecture, performance analysis, design verification, and electronic design automation. He was the founder of Chronologic Simulation in 1991 and was the principal architect of VCS, the Verilog Compiled Simulator. He was a co-founder of C2 Design Automation, now Forte Design Systems, where he continues to serve as Chief Technical Officer. He has 15 publications and 1 patent, and authored the Verilog Online Training course.

Florian Schäfer received his Ph.D. in Physics from Albert Ludwigs University in Freiburg, Germany in 1995, working on GaAs microstrip detectors and readout electronics. After 1 year at the École Polytechnique in Paris, he joined the Electronics Department at the international research institute GSI, in Darmstadt, Germany. He then contributed to the first DVD chipset developed at Thomson Multi Media. Since 2001 he has been working in Cadence's Methodology Service team with a focus on functional verification, SystemC, and ESL methodology.

This page intentionally left blank

CHAPTER 1

WHAT IS ESL?

1.1 SO, WHAT IS ESL?

The definitions of ESL—Electronic System-Level design—a successor to the venerable and still-used term *System-Level Design* (SLD), are numerous and confusing. For example, a July 5, 2006 search of Wikipedia [Wikipedia 2006] defines ESL as:

> Electronic System Level design, or "ESL," is an emerging electronic design methodology which focuses on the higher abstraction level concerns first and foremost.
>
> Electronic System Level is now an established approach at most of the world's leading System-on-a-chip (SoC) design companies, and is being used increasingly in system design. From its genesis as an algorithm modeling methodology with "no links to implementation," ESL is evolving into a set of complementary methodologies that enable embedded system design, verification, and debugging through to the hardware and software implementation of custom SoC, system-on-FPGA, system-on-board, and entire multi-board systems.
>
> ESL can be accomplished through the use of SystemC as an abstract modeling language.

Interestingly, Wikipedia does not have any definition of "system-level design," thus demonstrating a kind of Gresham's Law—new terms driving out old ones. Dataquest now positions "system-level," renamed "system design automation" (SDA) at the top of the design and verification abstraction chain, one step above ESL [Smith 2006]. However, the authors believe that it should still be included as part of ESL.

The term *ESL* originated with Gary Smith, until recently with Gartner/Dataquest and now an independent analyst, as a replacement for a variety of terms, including SLD and the earlier term Electronic System Level Design Automation (ESDA), which was coined in the 1990s (but never really caught on). Before Gary Smith joined Dataquest, Doug Fairbairn, President of Redwood Design Automation, and Ron Collett came up with a design abstraction taxonomy. They invented the term "electronic system design automation" (ESDA), which evolved to "system design automation" (SDA).

Later, Dan Skilkin, an EDA marketing manager, became the first head of marketing at Summit Design. He started calling his decidedly RTL tools "ESDA tools" and convinced the market that "ESDA" is equivalent to "graphical entry." In response, in 1995 Gary Smith substituted the term "electronic system level" for "electronic system design automation."

ESL, however, seems to have become part of the design and EDA industry's *lingua franca* fairly quickly. In the latest Gartner/Dataquest report on electronic design automation from December, 2005 [Smith 2005], ESL is defined as "concurrent design of hardware and software." The report then goes on to define a behavioral level ("ESL design prior to hardware/software partitioning"), an architectural level ("ESL design after hardware/software partitioning"), which consists of architectural design ("ESL language-based design using synthesis technology to output an RTL [Register Transfer Language] design description"), and platform-based design ("ESL-model-based design using an existing platform and mapping technology to output an RTL design description"). The architectural level by the Gartner definitions seems to leave out software specification, design, and implementation. Gartner further defines 35 subcategories of ESL, covering areas such as algorithmic design, ESL verification, processor/memory power analysis, control logic design and analysis, and so forth.

The problem with ESL is manifested by both of these definitions. ESL seems to be a collection of many different activities and methodologies for designing "systems" of various types. These "systems" are electronic based (as opposed to optical, mechanical, hybrid, or other kinds of systems), yet they involve both hardware and software. Clearly, there is something abstract about ESL—at least, more abstract than the traditional RTL- and programming language (e.g., C, C++, Java)-based hardware and software design methodologies in widest use.

In 2004, Tets Maniwa wrestled with this same conundrum [Maniwa 2004]:

> What is ESL? Unfortunately, in the rapidly growing area of design tools, getting everyone to agree on basic definitions is a challenge. The definition of ESL is in a state of flux. Originally, the term referred to the concept of system-level design. Now, ESL seems to include at least some reference to hardware–software interactions, as well as higher levels of abstraction. A possible distinction is to see ESL as an activity, not as a language syntax. Either way, ESL needs to address the system-level tasks.

This seems to cover the same ground as the previous two definitions, and leaves us in much the same quandary.

Taking a more colloquial angle, one of the authors heard a definition many years ago from Pierre Paulin, now of STMicroelectronics: "System-level design is design at a level one level above the one you are currently designing at" (Pierre may have coined this definition, or may have passed it on, its origins lost in the mists of time). The nice thing about this definition is that it represents both the abstraction and the ephemeral nature of ESL, sometimes conceptualized using the story of the blind

men and the elephant, or described as an attempt to grab hold of a cloud or embrace a marshmallow.

Taking all these approaches into account, the authors have come up with a working definition of ESL as "the utilization of appropriate abstractions in order to increase comprehension about a system, and to enhance the probability of a successful implementation of functionality in a cost-effective manner, while meeting necessary constraints." This combines the ideas of system, abstraction, and (implicitly) concurrent specification—the design of hardware and software—in that it talks about "implementation of functionality" without regard for whether that is to be done in hardware, software, or a combination. It also implies a process (where comprehension of the system is increased by using abstractions, and this has a downstream beneficial effect on implementation), some level of system optimization (cost-effectiveness), and proven benefit (appropriate abstractions). This is not a bad starting point for a definition of ESL, given its multifaceted and all-encompassing nature.

Probably, we will arrive at a truly satisfactory definition of "ESL" only some years after the majority of design teams have started to use some aspects of an ESL methodology. At that point, the various definitions will have coalesced into an industry consensus as to what ESL is all about. Also at that point, it will seem blindingly obvious to all of us (the blind men trying to deduce an elephant), and we will wonder about all the past confusion.

In the meantime, we'll stick with our foregoing definition. The key things to remember about ESL are the words "system," "abstraction," and "process." This book, *ESL Design and Verification: A Prescription for Electronic System-Level Methodology*, gives a snapshot of the state-of-the-art with ESL in mid- to late-2006. We believe that the art and science of ESL-based design and verification has advanced to the point where it can be summarized in a book such as the one you are reading. We also believe that enough is known about the various topics and methods in ESL methodologies that we can give some very pragmatic advice about the approaches that should be used, as well as those approaches that may not be ready for use or may indeed be unsuitable. Hence, the subtitle for our book reflects the fact that we try to be "prescriptive" about the methodology whenever possible. The book title also reflects the fact that although much of ESL has been focused on design aspects up to the present, and although much of the commercial Electronic Design Automation (EDA) tool revenue in the ESL domain comes from verification tools, a holistic view of ESL must embrace both design and verification aspects—all driven from a specification that tries to capture both the functional intent of a system and the constraints under which it must operate, in order to yield an optimal or near-optimal result by the time implementation has been completed.

One point must be dealt with early on: ESL methodologies have many starting points, and there is no "right one" for all designs in all design domains. In addition, there are many specification, design and verification languages, and notations that could be used in an ESL methodology and design flow, and there is no "right one" or even a "right set" of such languages. What is important to recognize is

that there are "right principles" in an ESL methodology, and these may be implemented in flows incorporating multiple languages and notations. Naturally, in a book of this type, the authors may illustrate points using some specific languages, and will definitely have opinions on the right use models for various notations. But successful ESL methodologies may be implemented using a wide variety of languages, and those discussed here in detail should be thought of as merely illustrative examples.

Just as there are many starting points, partly dependent on the design domain in which one is working (e.g., heavily control dominated vs. dataflow, or combinations of both), there are many ending points. One can successfully take a product concept into an implementation involving lots of new hardware and software design; heavy reuse (using a platform-based design concept); very little new hardware but lots of new software, or vice versa; intensive development of new algorithms for a design domain and their implementation in software, hardware, or both; configuration and extension of processor Intellectual Property (IP) or the use of standard fixed processors, or a mixture; and many other hardware and software architectures.

The flow can also be top-down (e.g., algorithm-down), bottom-up (driven from a platform for a derivative design), or, much more likely, "middle-out" (a mixture of bottom-up and top-down for new or substantially modified functionality). Flows will depend partly on starting and end points, and on languages and notations, as well as on application and design style and domain. As a result, there can never be just one ESL flow, and never just one "correct" interpretation of how ESL methods, models, and tools can be used.

Nevertheless, the state of activity in all aspects of the ESL community—system designers, semiconductor designers, platform providers, IP block providers, tool vendors, modeling companies, consultants, academic researchers, and, indeed, book authors!—means that the time is right for a book on ESL methodology, and this is that book. We don't anticipate that this book is the final word on ESL—not by a long shot. However, it may serve as an adequate starting point for anyone interested in ESL-based design tools, methodologies, models, and flows. And by offering our modest prescriptions on the best ESL methodologies to use, we hope that readers will be able to translate the descriptive part of the book into actionable steps they can take to improve the design processes and results for all the projects they are or will be working on.

The move to ESL design and verification is a fundamental shift in design methodology. It offers measurable improvements in design productivity, design quality, and reduction of risk in product development. There are no other shifts in the design process that hold as much promise and demonstrated results in better meeting design objectives than ESL. The time to begin moving to incorporate it into your design flow is now. As the reader will see throughout the rest of the book, there are practical concrete steps that can be taken by any design team to incorporate ESL methods, models, and tools into their own flow. Your investment in the time it will take to read on to the end will be well-rewarded.

1.2 WHO SHOULD READ THIS BOOK

This book has been written for many kinds of readers. We have endeavored to bring together the knowledge, insight, and foresight of a large number of expert contributors in all aspects of ESL design and verification, and the three authors have worked to knit the book together into a consistent flow that treats all aspects of ESL design. The next section describes the structure of the book and gives advice on reading it. However, we believe that all will benefit from the book in the following ways:

- The novice in ESL, who may be a student in electronics design, EDA, software design, system design, or related fields of study, or may be a designer seeking to learn more about ESL design and verification, will find an excellent overview of the topics involved in ESL. By presenting an ESL design and verification flow, preceded by sections about the history of ESL and the key enabling factors in establishing this area as an important part of design, we give insight into all phases of the flow, which is especially important to the novice. The definitions and taxonomy chapter will be useful to the novice in understanding the vocabulary and language spoken in the ESL domain. The overviews in each chapter and extensive references give plenty of scope for additional learning.
- The experienced designer who has some base in ESL design may find that there are many aspects of the process with which he or she is not familiar. For example, an algorithm specialist, who is familiar with specifications and prepartitioning analysis, may be unfamiliar with many aspects of the downstream verification, analysis, and implementation processes. A hardware implementation expert may not know very much about the software processes beyond the defined hardware–software interfaces. A software development expert may wonder about the algorithmic specification they are implementing, and how it can be analyzed before detailed implementation proceeds. A verification engineer may wish to understand how their discipline can move up to the ES level, and the different kinds of issues that may be important there. This book provides a good opportunity for specialists in all aspects of the design flow to fill in gaps in their experience and knowledge base and improve their understanding of the overall tradeoffs and issues.
- Design managers who wish to understand the ESL flow and particular parts of it to improve the capabilities of their design team; want to reduce the risk in their development projects; want to expand the scope of their design projects and identify key missing components of the team; or just wish a greater overview of the whole of ESL to improve development processes and project planning, will have a lot to gain by reading this book from cover to cover. The chapter on future research possibilities, as well as the "provocative thoughts" sections found in many of the chapters, will alert them to areas they should watch for in the future. The "prescriptions" found in each chapter represent a good checklist of design and verification practices that they can use to measure against their own team.

- Researchers in academia or an industrial research laboratory may have an excellent view of their subject area, but not have a good view of how it fits into the overall ESL design space. In addition, they may not be fully aware of the state-of-the-art industrial practice in ESL, those methods that have been proven to work and are recommended, and those methods that have not found favor with practical design teams. This book will give them considerable insight in particular into industrial practice. In addition, areas that are subjects of research for only a few specialists, such as ESL verification, are treated at some length here, and this may open up opportunities for future research. We hope as well that the taxonomy defined in this book will be adopted by the academic community, and extended or modified with the advancements they are making. A common classification scheme will help remove a lot of the ambiguity that has been so prevalent in this field. Finally, the discussion in the future research chapter may be familiar to most, but may contain some new thoughts for some in the research community, and again open up new opportunities for effective, industrially relevant research.

1.3 STRUCTURE OF THE BOOK AND HOW TO READ IT

The book is divided into three major sections:

- The first section contains general overview material, including the introduction, taxonomy of ESL and definitions, evolution of ESL, and a discussion of key enabling factors that demonstrates that *now* is the time to adopt ESL design and verification flows, and that illustrate the prerequisites for successful adoption.
- The second section is the heart of the book. It contains several chapters that give, first, an overview of an idealized ESL flow, and then step through each stage of the flow, from specification creation, pre-partitioning analysis, partitioning, post-partitioning analysis, post-partitioning verification, hardware implementation, software implementation, and, finally, implementation verification.
- The final section contains one chapter on emerging and future research in the ESL area.

The first section will be useful for everyone because it provides a valuable context for ESL. The second section will be useful to everyone wanting details on the flow. The third section may be of most interest to researchers, students looking for research topics, and managers who want to know what may be coming next.

Each chapter concludes in general with three sections. The first is a "Provocative Thoughts" section, which is meant to trigger new ideas, stimulate thinking in new directions, and break the mold (or get out of the box) of current methods and tools. The second is the summary of what has been covered in the chapter. The final section,

"The Prescription," is a summary of the key lessons of the chapter presented as a set of strong recommendations or guidelines to follow in implementing a state-of-the-art ESL design flow.

1.4 CHAPTER LISTING

- Chapter 2, *Taxonomy and Definitions for the Electronic System Level*, classifies ESL models, methods, tools, and approaches using a standardized set of classification axes. It also gives definitions of terms to be found later in the book. The acronyms used throughout the book are defined in a list at the back of the book.
- Chapter 3, *Evolution of ESL Development*, discusses the history of ESL design and verification, a number of the earlier tools and projects from the previous generations of ESL, and discusses many of the factors that make ESL particularly relevant as a methodology in 2006 and 2007.
- Chapter 4, *What Are the Enablers of ESL?*, discusses factors in the composition of the design, tools, and IP industry that are necessary prerequisites for ESL-based methodologies, their status as of today, the important roles of standards organization, and the place of open source technology as a key enabler for ESL design.
- Chapter 5, *ESL Flow*, is a linchpin chapter that opens the part of the book dealing with the detailed ESL flow. It contains summaries of the succeeding chapters, 6 through 13, that provide the background and details on each part of the ESL flow methodology. Chapter 5 should be read as an overview before diving into all or any part of Chapters 6 through 13. It is also a useful reference for anyone wanting to get an overview of ESL methodology.
- Chapter 6, *Specifications and Modeling*, discusses management of ambiguity, design languages, and notations important for capturing system specifications, and outlines the important role that a requirements management system can play in an ESL design flow. The state of a number of current and future notations is discussed and recommendations given.
- Chapter 7, *Pre-Partitioning Analysis*, covers the kinds of static and dynamic analyses that are possible with the specifications developed in Chapter 6. These allow early estimates of design performance, power consumption, cost, and development effort to be prepared. They are also used to set performance envelopes and system constraints, and to characterize fundamental operating conditions for the algorithms that a system may implement.
- Chapter 8, *Partitioning*, discusses the partitioning of a system description and function into hardware–software architectures and, in the case of multiprocessor systems, software–software architectures. It outlines the issues of mapping system functions onto the architectural elements, validating the partitioning and exploring the possible design space for system implementation.

- Chapter 9, *Post-Partitioning Analysis and Debug*, describes the kinds of analyses that are possible once a system has been decomposed into its major architectural elements, and the ways in which tools, models, and methods interact to allow the system to be analyzed in detail. Some of these analyses might cause a change in partitioning or choice of system components, in which case iterations back to partitioning might be needed.
- Chapter 10, *Post-Partitioning Verification*, describes the beginning of the verification process for hardware and software that should occur once partitioning has been completed. Although final implementation has not yet occurred at this phase, the partitioning process and component selection have refined much of the design, finalizing many design requirements. It is important to begin the verification process early, guided by the design specifications and the system characteristics at this point.
- Chapter 11, *Hardware Implementation*, discusses the various ways in which functionality destined to be implemented as hardware components might be implemented. A survey of different methods is given, before focusing in detail on the capabilities of new and recently emerging ESL synthesis tools that are becoming more important in the overall ESL flow.
- Chapter 12, *Software Implementation*, describes the intersection between ESL design flows and traditional and more recent software implementation and validation methods. In particular, the two aspects of driving the creation of ESL software from ESL specifications, and using ESL models as part of the development environment for ESL software, are discussed in detail, along with relevant approaches for debugging system software using models and tools.
- Chapter 13, *Use of ESL for Implementation Verification*, discusses how the design flow from ESL allows the development of more effective verification environments for the implementation stage. In particular, use of the right kind of ESL models will improve functional coverage and make stimulus generation easier. The models developed earlier need to be modified to incorporate additional architectural and microarchitectural detail to best drive this stage of the process.
- Finally, Chapter 14, *Research, Emerging and Future Prospects*, discusses some of the current, leading-edge, system-level design projects, and what they may teach us about the future evolution of ESL methodologies. Some future architecture trends, including the rise of multiprocessing, are elaborated. It also discusses other areas of evolution in ESL, including the impacts of globalized development, value migration in the ESL flow, educational requirements for the next generation of designers, and the future of the commercial ESL tools industry.

We hope that all who read this book will find at least one key idea expressed herein to be of value—and that most will find more than one!

1.5 THE PRESCRIPTION

1. Many facets of ESL are real and useable today, even if they are not being handled or automated by tools. We highly encourage all companies to evaluate the use of these or similar proven techniques, and to start planning for their future incorporation as a major part of their system design flow.
2. We believe that the existing RTL methodologies are no longer scaling and that system complexity needs to be handled in a new and more effective manner.
3. It does not matter if you are a hardware, software, or system designer, verification engineer, tool vendor, or IP supplier: only through talking to all of your suppliers, partners, or customers about ESL will a full comprehension of the community's needs be gained. Partnerships are stronger than individual efforts.

References

R1.1—So, What Is ESL?

[Maniwa 2004] T. Maniwa, Focus report: Electronic system-level (ESL) tools: A bolt-on to RTL or a new methodology?, *Chip Design Magazine*, April-May, 2004, pp. 17–21.

[Smith 2005] G. Smith, D. Nadamuni, L. Balch, and N. Wu, Market trends: Electronic design automation, worldwide, 2005, Gartner/Dataquest, 5 December 2005, ID No. G00136302.

[Smith 2006] Gary Smith, private communication to Andrew Piziali, September 25, 2006.

[Wikipedia 2006] Electronic system level; available at http://en.wikipedia.org/wiki/Electronic_system_level. Accessed July 5, 2006.

This page intentionally left blank

CHAPTER 2

TAXONOMY AND DEFINITIONS FOR THE ELECTRONIC SYSTEM LEVEL

When standards or industry groups first get together, they often have the problem that each participant has their own words, phrases, and terms that they use to explain their ideas or concepts (while of course acknowledging that there are also "political" agendas that some advance in standards bodies using certain terms and language). As a result, no one agrees with each other's opinions until they realize that the definitions they are using are the problem. Once they have clarified those definitions, they may then discover that they all agree with each other and the only thing left to do is to agree on the terms and their corresponding definitions. This chapter attempts to mitigate these problems by defining a set of terms, definitions, and a classification system that ensures everyone knows what we are talking about in this book. In addition, because this is the first book in this technology space and tool support is just becoming available, it may be possible to establish a baseline for the whole industry to rally around.

The chapter is divided into two main sections. First, a taxonomy for the ESL space is defined. This is an extension and adaptation of an existing model taxonomy. The model taxonomy defines what we mean by *abstraction*, and the ESL taxonomy defines the factors that differentiate parts of the complete ESL landscape. We then provide a set of definitions for the terms used in this book, along with cross-references to their use. This provides context for many of those definitions. The acronyms used in the book are defined and found in a list at the back of this book.

2.1 TAXONOMY

2.1.1 Introduction

A *taxonomy* is a characterization of objects or concepts based on the relationships that exist between them. A taxonomy can be represented in a hierarchical graph or table of attributes, where each of the attributes identifies a particular element of

the differentiation. The title of "Father of Taxonomy" is given to Carl Linnaeus, a Swedish scientist who provided the characterization of living things in 1735. His taxonomy concentrated on the reproductive organs of plants and animals, and although many modifications have been made to it since then, the core taxonomy of living things in use today remains true to his concepts.

In 1869, with the development of the periodic table of elements, a second important taxonomy came into existence. Dmitri Mendeleev noticed that if all known elements were arranged in order of their atomic weights, a repetition of properties was observed. The periodic taxonomy of elements enabled him to identify elements whose atomic weights were in his view incorrect, and was so powerful that it allowed the prediction of other elements that had not yet been discovered.

Creating a taxonomy for ESL deals with somewhat of a moving target because the parameters change as the technology matures. This can introduce some controversy, but we try to categorize elements of the ESL design flow in a way that may be useful for defining and constructing design flows. Although we have attempted to categorize according to properties that are as independent as possible, complete independence has not been achieved.

2.1.2 Model Taxonomy

The ESL taxonomy is based on the model taxonomy, discussed in the following section, that was first developed by the Rapid prototyping of Application Specific Signal Processors (RASSP) program. In 1995, the Terminology Working Group of the RASSP program began the definition of a common set of model properties. The intent was that models to be interchanged among the participating companies would be characterized according to these properties so that the receiver of each model would know exactly what each model contained. With the completion of this program, the model taxonomy was transferred to the Virtual Socket Interface Alliance (VSIA), where the System Level Working Group continued development and released their first version in 1998. After a number of iterations and the creation of three new taxonomies covering verification, platform-based design, and hardware-dependent software, all four taxonomies were transferred to three editors—Brian Bailey, Grant Martin, and Thomas Anderson—who merged those four taxonomies into a single consistent work and updated them with recent developments. The resulting work was published in 2005 [Bailey 2005]. The reader should consult that work for a full description of the model taxonomy.

The model taxonomy is composed of four main axes, each of which is described briefly. It also introduces a fifth axis, the software programming level, but that is not discussed in the context of this book. The four axes are temporal, data, functional, and structural. These four axes are not completely orthogonal, and the functionality axis reflects to some extent the concepts of temporal and data abstraction. In addition, the model taxonomy provides both an internal and external definition for each model, so that the abstraction of the model internals may be different from the abstraction shown on its interface. A typical example of this is an instruction

set simulator (ISS). Internal to the model, a high degree of data and timing abstraction is used, and will probably bear no resemblance to the internal structure of the processor beyond perhaps a pipeline model. However, it is possible that the ISS will be connected to a Bus Functional Model (BFM) that will convert the abstract function calls that correspond to bus accesses, transforming them into a pin accurate model that can be instantiated directly into a design. In this case, the internal and external abstractions of the model are very different.

2.1.2.1 Temporal Axis

The temporal axis defines the timing in the model. The defined points are "partially ordered events," "system event," "token cycle," "instruction cycle," "cycle-approximate," "cycle-accurate," and "gate propagation accurate." Not all points are described here. Aspects of this axis are carried forward into the ESL taxonomy.

The most abstract point defined on the scale is partially ordered events. This means that we know when something will start and finish only in terms of its relationship to when other things start and finish in this particular execution. We cannot place any notion of actual times on these events. Further down the scale is token cycle accurate. This is typical in dataflow systems where a data arrival clock can be thought of as a regularly scheduled event. The instruction cycle, cycle-approximate, and cycle-accurate points all have the notion of an actual clock with a known period, and thus actual times are known. However, different levels of accuracy are still possible because the exact number of clock cycles that elapse between significant events has some degree of uncertainty. At instruction cycle accuracy, no account is usually made of wait states or the impact that hardware may have on the execution times of software. In fact, the memory accesses normally happen only in a virtual sense. At cycle-approximate accuracy, the actual operations of the bus or memory accesses are present, but the timing between them is not known precisely. At the cycle-accurate point on the axis, exact cycle counts are known. Gate propagation accuracy is the final point on the axis at which the timing within the clock period is also known precisely.

2.1.2.2 Data Axis

The data axis defines the level of precision of data. Fewer points are defined on this axis than on the temporal axis. They are "token," "property," "value," "format," and "bit logical." A token indicates that some data is moving through a system, but its size and content are unknown. A property may be an enumerated type where a name is given to something, even though how this name will be represented is not defined. Value could be an integer, floating-point value, or similar data type, where its numerical accuracy is known but how it is represented in hardware is not known. Format is a processor-like data format, dealing with issues such as endianess, and fixed- vs. floating-point. Bit logical provides that mapping of value into the hardware that will store it. Aspects of this axis are carried forward into the ESL taxonomy.

2.1.2.3 Functionality Axis

This axis attempts to define the precision of the operations themselves. Only three points are defined: "mathematical relationship," "algorithm," and "digital logic/Boolean operation." The concept here is that a mathematical relationship defines precedence but not order. We may select an algorithm to implement that relationship which defines the order in which things are done; eventually, the digital logic/Boolean implementation model instantiates the functional units that will be used to implement that algorithm. This axis is not truly independent of the others because it conveys notions of timing and data resolution as well as structure. It is somewhat superseded by the ESL taxonomy presented in the next section so no aspects of this axis are carried forward into the ESL taxonomy.

2.1.2.4 Structural Axis

This axis attempts to convey the amount of structural detail present in the model and thus how close it is to the actual implementation. It is an absolute scale in that it does not provide a way to see if a model has a structure that is different from the final implementation structure. The model taxonomy suggests that if a block is used that does not conform to the final structure of the implementation, it is considered a black box having no internal structure. However, in this book, the term "opaque box" will be used rather than black box, to refer to this concept. No aspect of this axis is carried forward into the ESL taxonomy.

In the ESL taxonomy presented in the next section, the temporal and data axes of the model taxonomy are reused directly, but the functional and structural axes are not used because they do not provide any useful attributes that help us to distinguish tools or flows in this emerging space.

2.1.3 ESL Taxonomy

This section introduces the three new axes of the ESL taxonomy, namely concurrency, communication, and configurability. The role of each and the ways in which they interact are discussed. The three new axes coupled with the two axes preserved from the model taxonomy, temporal and data resolution, are used to show how language attributes, tools, and flows can be defined.

Although we have defined Electronic System Level (ESL) design more formally in Chapter 1, people often informally think of ESL as being the design of systems including a mixture of hardware and software. Most companies in this space today start with a description embodied in a software-executable specification. This description may be at a number of different levels of abstraction, such as a pure algorithmic description or perhaps incorporating some architectural decisions, but it is built with the expectation that it will execute on a processor. This starting point is already in the form of an implementation solution. Tools in the ESL space map all or part of the software into hardware of various kinds, or may leave the application in its entirety on an implied processor or set of processors, with the tool making these decisions explicit. A wide variety of solutions is possible, ranging

from a single processor executing the code, to an all-hardware implementation, to a solution where the application is mapped onto multiple processors. Whenever the solution is divided among multiple execution engines, there is the possibility for concurrent execution, and concurrency is the first axis of the ESL taxonomy. In addition, it is likely that some form of communication will be needed between the threads, unless they are actually completely independent threads of execution. Thus, communication is the second axis of the ESL taxonomy. These two and the interaction between them are described in the next two sections.

2.1.3.1 Concurrency

Concurrency defines the amount of processing or execution of an application that can be performed simultaneously. An implementation does not have to utilize all of the available concurrency because this may produce a solution that does not fit the nonfunctional requirements of the design, such as cost, size, or power consumption constraints. In addition, more concurrency increases the difficulty of verifying an implementation.

Refinement of a system design includes the definition of both programmable and fixed-function architectures consisting of a number of execution resources. Indeed, this refinement typically occurs a number of times at different levels of granularity. Introduction of separate execution resources creates the opportunity for concurrent execution of parts of the system functionality.

The ESL design space and solution space are very diverse. An ESL flow may start from a description of the design that is at a certain level of abstraction, or particular computation model. For example, a design described in the C or C++ languages is a sequential model that describes a solution capable of running on a processor. Some companies and standards groups have created language extensions that either allow hardware concepts to be included in those languages, such as SystemC, or add some specific capabilities such as defining possible concurrency, as is done in HandelC.

Alternatively, an algorithmic model may have been developed that makes no such assumption about the underlying implementation and may have concurrency inherently built into the description. An example is a mathematical equation, where dependencies for calculation are directly built into the expression itself. However, although they define dependency, they do not explicitly define order. It can thus be seen that this axis in part replaces the functionality axis of the older model taxonomy, but does so in a way that defines a specific characteristic of the model rather than a vague notion of functional abstraction.

Given that a software implementation is likely to be a sequential solution (ignoring very coarse-grained concurrency defined by threads or some other operating system-level mechanism) and a hardware implementation more concurrent, it is inappropriate to define which of these models is at a higher level of abstraction. They are just different. A transformation step would be necessary to convert from the algorithmic to the C model, and a similarly difficult transformation exists in the opposite direction.

The ESL design space encompasses tools that automate, or provide assistance to the traditionally manual process of synthesizing an architecture consisting of multiple execution resources and assigning the parts of the system functionality to these resources. In general, synthesis is a process in which resources are created and operations are scheduled and mapped onto those resources. In that respect it creates and manages concurrency. ESL synthesis, then, is the creation and management of concurrency in systems consisting of hardware and software. It is clear then that concurrency is a central axis in both the input to ESL and the result from it.

The discrete points on the axis are defined as:

1. Sequential (none) — No explicit concurrency exists in the function. This does not mean that it is not possible to extract concurrency from the function by looking at control or data dependencies, but that no indications exist in the function to identify such concurrency. An example of this would be ANSI C descriptions.
2. Multi-application — The concurrency that exists here is very coarse grained and not built into the operating characteristics of the system directly. In other words, two independent functions may share some data contained in a file, but one function has no direct influence on the other. An example of this would be a Microsoft Word document and an Excel spreadsheet, where it is possible to embed data or charts from the spreadsheet into the document. Whenever the document is opened, the updated chart or data from the spreadsheet would be imported.
3. Parallel — This level of concurrency concerns multiple functions that are operating independently, but may need to cooperate for certain activities. An example is a transaction processing system where data is explicitly shared between each function or instance of the function. Data locking on those accesses has a direct impact on the running of each function and may trigger further functions to be executed. Normally, one would not be able to predict when such synchronization points would happen because the events that initiate them are independent.
4. Multithread — A single function may have been defined with multiple threads of execution. This concurrency is thus explicitly built into the operation of the function and may have explicitly defined synchronization points. The means of synchronization and data transfer are built into the function. Examples of this are plentiful in both the hardware and software domains, ranging from the operating system running on a computer to multiple pieces of a hardware system that have divided up the task into independent functioning blocks. This is common in cell phones where a Digital Signal Processor (DSP) would be running a wireless decode algorithm at the same time a general-purpose processor is controlling the call.
5. Pipeline — In general, pipelining defines the streaming of data from one operation to the next in a controlled manner. This also highlights the hierarchical

nature of a system, such that within a coarse pipeline stage, we could consider a finer level of parallelism and communications also implemented as a pipeline. There is a range of different granularities when we talk about pipelines. At the coarsest level, there could be large blocks of computation that could be executed within a dedicated processor, programmable coprocessor, or fixed-function custom hardware block, and the results of this may be fed to another, similar subsystem. Examples of these kinds of pipeline stages could be a Fast Fourier Transform (FFT) or multitap filter. At a finer level, pipelining is very common in processors where the main data path will be constructed as a pipeline, with stages such as instruction fetch, decode, argument fetch, execution, and writeback. It could also be considered to be the default level of definition for an RTL description, where the combinatorial actions to be performed between registers are defined.

6. Signal level — This is a very fine level of concurrency that is typical in a gate-level or asynchronous description. Multiple combinatorial paths exist between two synchronization points and signal transitions progress through the system as fast as the circuitry will allow. This may create transient values along the processing chain and would normally be combined with some degree of pipelined design so that synchronization points can be defined.

2.1.3.2 Communication

When more than one processing element exists, there must be communication between them. That communication can take many forms and is highly dependent on the architecture of the final solution. Very fine-grained parallelisms, such as operations within an instruction or dedicated pieces of hardware, are naturally handled by point-to-point communication or pipelines. At the other end of the spectrum, two software threads that need to communicate are likely to use some form of shared memory to communicate with one another. For example, an all-software solution could use shared memory, or pipes layered on top of shared memory. A multiprocessor software solution could use hardware FIFOs (First In, First Out) or queues between processors. An architecture that deploys a processor (or multiple processors) and programmable coprocessor(s) might use a coprocessor interface, or the coprocessor can have its own Direct Memory Access (DMA) interface. When software and hardware communicate they could also be using shared memory, but the hardware could be tied very closely to the processor as a coprocessor. Communication between dedicated hardware elements could be through dedicated structures such as FIFOs or the register stages of a pipeline. These examples show that although concurrency and communication are primarily orthogonal to each other, there are many points in the total solution space that do not make a lot of sense on the basis of economic or performance concerns. The discrete points on the axis are defined as follows:

1. Point-to-point — This allows two concurrent functions to communicate with each other directly without any additional form of control between them.

This would generally be found only in hardware solutions because it implies the continuous flow of information rather than something that is intermittent, periodic or not.

2. Buffered (also includes FIFO) — This is the most elemental form of controlled communications that can be made between two concurrent tasks. It allows for the execution rates of two functions to be different and thus provides some degree of electrical and timing isolation between them. If the buffer depth is greater than one, as would be the case with a FIFO, then rate independence can also possibly be assured, at least to the depth of the FIFO.

3. Coprocessor — Although this may sound like a processor-dependent term, it means that the information processed across the two functions is being shared in some manner. It is likely that one of the functions is the owner of the information and may have to do some work in order to make it available to the other function. In other words, a producer–consumer relationship exists between them. In the case of a processor/coprocessor pair, the processor would own the information, but at the time of the request the information may be in a register, cache, or memory. The coprocessor is unaware of its actual location, but it does affect the rate at which the information is returned. A coprocessor can be programmable or fixed-function hardware.

4. Memory — Memory is a very common way to transfer information between two functions. At this level, it is assumed to be multiport such that each function can read or write to the memory independently. It would also imply that they have a dedicated connection to the memory and not have to access it through a bus mechanism (see the next point on the scale). Examples of this often occur in a client–server type of relationship, where multiple clients may look for transactions placed in the memory that they are able to perform and extract the necessary data. This is unlike the buffered form of communications, where the data is ordered and transferred point to point.

5. Bus based (high speed) — In software systems where the function is run on a processor, almost all communications are made through one or more buses, each of which may have different transfer rates or latencies. Several buses may be used to make the necessary transfer, with bus bridges mapping the requests between them. This category is meant to include all buses directly associated with the processor and does not, for example, make the distinction between the Advanced Microcontroller Bus Architecture (AMBA), Advanced eXtensible Interface (AXI), and AMBA Advanced Peripheral Bus (APB) buses of an ARM processor. This category would also contain the emerging architectures called Network On Chip (NoC). When a communication request is made between any two of the processors, the routing resources within each of the processing elements act as a bridge between each of the processor buses.

6. Bus based (low speed) — In this context, a low-speed bus means that there is a more extensive protocol being used on the bus to ensure transport integrity, or that is capable of allowing things to be connected at much larger distances.

Examples of this within a computer world would be a Peripheral Component Interconnect (PCI) bus or a Universal Serial Bus (USB). It could also include an Internet connection where functions would be communicating over great distances and have very long latencies.

7. None — If the functions do not need to communicate, then no channel is necessary for their communication. This also implies that the concurrency between the functions does not create any issues with which the system designer must be concerned.

In a typical ESL solution, there may be several types of concurrency present at the same time. For example, a multiprocessor solution may be defined with each processor communicating through a global memory area, but at the same time, each processor may have coprocessors that communicate through an internal processor interface and have an instruction pipeline, where buffered communications occur between the stages. Different tools will probably concentrate on particular types or regions of concurrency, but there is nothing to say that future tools may not be able to handle a broader range of concurrency types.

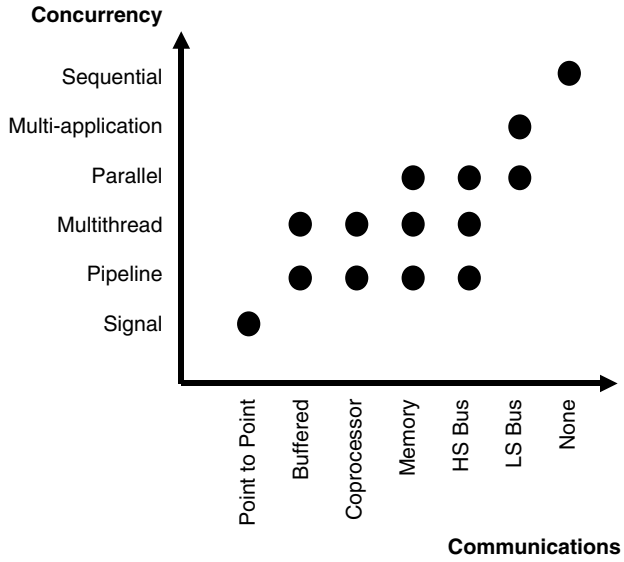
2.1.3.2.1 Concurrency and Communications

The way in which the two axes, concurrency and communication, interact is interesting because it defines the envelope of technology solutions that have so far been attempted, trading off granularity of concurrency and speed of communications.

Although it has already been stated that in an ideal world the axes would be completely orthogonal, there are at times other factors that limit the range of choices. These can be cost, performance, and power consumption, among others. This can be seen clearly by looking at these two axes together. Figure 2.1 shows some of the more common points on these axes where solutions exist. As can be seen, not all points are filled in because the coarser levels of concurrency can afford to use more generic, shared forms of communications, whereas very fine levels of concurrency need more dedicated, faster access methods.

2.1.3.3 Configurability

The story does not end with concurrency and communication: there is another important axis necessary to define ESL solutions. That axis is the configurability of the solution. Most people would consider software to be configurable, but the configurability of the platform on which the software runs, or in which it is embedded, is just as important. The only difference between hardware and software in this regard is the delivery mechanism. Software, written in such languages as C or C++, is modified at design time and compiled to work on existing hardware solutions, but once it is shipped it cannot be easily modified in the field, although mechanisms may exist on some devices for downloading new or modified compiled executables from a network. Java code, on the other hand, can be modified or retargeted for different hardware because the compilation step is deferred until close to execution time and thus has more configurability.



■ **FIGURE 2.1**

Concurrency and communications.

Dedicated fixed-function hardware solutions are often very rigid and can perform only a single task. A processor with a fixed instruction set is also considered a fixed solution. However, a processor with an extensible instruction set has some degree of configurability. This configurability is handled by the tools in the design path. Hardware can also be built to be configurable. For example, a Serializer/Deserializer (SerDes) I/O block can be configured to support a number of different serial protocols. Many communications processors can also support the higher levels of the protocol for these communications mechanisms. A reprogrammable fabric, such as an FPGA or Programmable Logic Device (PLD), can be reprogrammed at start-up, or even possibly reprogrammed during operation of the system. Although run-time reconfiguration is not in common use today, there is no reason to think fully dynamically reprogrammable systems will not exist in the future that can change personalities to configure themselves for the task or computation loads that they face at any given time. In fact, the history of computing shows such run-time reconfiguration in a few examples, such as the Burroughs B1700/1800 that reconfigured its instruction set depending on the target language of the program being run (e.g., FORTRAN or COBOL) by dynamically using a different set of microcode.

The defined points on this axis are as follows:

1. Fixed — The user of a fixed block or device has no ability to make any kind of changes to it. A fixed design may be provided at any level of abstraction, in hardware or software. It may be provided as a black box lacking any kind