Sandro Rigo · Rodolfo Azevedo
Luiz Santos *Editors*

Electronic System Level Design

An Open-Source Approach

Springer

Electronic System Level Design

Sandro Rigo • Rodolfo Azevedo • Luiz Santos
Editors

Electronic System Level Design

An Open-Source Approach

Springer

Editors
Sandro Rigo
Instituto de Computaçao
Universidade Estadual de Campinas
Av.Albert Einstein 1251
Campinas, São Paulo 13083-970
Brazil
sandro@ic.unicamp.br

Rodolfo Azevedo
Instituto de Computaçao
Universidade Estadual de Campinas
Av.Albert Einstein 1251
Campinas, São Paulo 13083-970
Brazil
rodolfo@ic.unicamp.br

Luiz Santos
Depto. Informática e Estatística
Universidade Federal de Santa Catarina
UFSC Campus
Florianópolis, Santa Catarina 88030-300
Brazil
santos@inf.ufsc.br

ISBN 978-1-4020-9939-7 e-ISBN 978-1-4020-9940-3
DOI 10.1007/978-1-4020-9940-3
Springer Dordrecht Heidelberg London New York

Library of Congress Control Number: 2011927759

Cover design: VTeX UAB, Lithuania

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Electronic System Level Design, or ESL Design, is generally understood as the set of tools, methodologies and design techniques applied to modern electronic systems design, from high-end chips and systems, to embedded devices, to integrated hardware and software systems. Given the complexity of current systems, advanced tools and methodologies have become absolutely essential to achieve the necessary productivity, quality, cost and performance expected in a design process. One of the important tenets in ESL Design is the need for early design analysis. This is done mainly through high-level modeling and simulation, performance and power analysis and functional verification, before committing the design to lower-levels of abstraction aimed at synthesis and optimization. This is especially true for complex systems involving different types of components such as processors, custom blocks and software. In fact, it may be totally impractical to simulate such systems at a low-level of representation such as register-transfer level, due to extremely long simulation times. High-level models are simpler to write, understand, optimize and debug than lower-level models, and they can simulate significantly faster. The more the design can be refined, optimized and verified at a high-level of abstraction, the higher the overall design productivity, the better the quality and consequently the lower the cost of the final result. However, high-level models and development environments are not without their own difficulties. It is not simple to write a high-level model at the appropriate abstraction level which will result in the best trade-off between architectural details and simulation speed. For this end, researchers have formalized different abstraction levels at different levels of architectural and timing accuracy. Depending on what types of design analysis need to be done, one abstraction level may be more suitable than another. This adds to the modeling complexity, and in the worst case, if multiple models need to be developed, it may start reducing the productivity advantage of a high-level modeling methodology. A high-level modeling and simulation framework is a complex software engineering challenge. Different types of models, such as a processor model, custom blocks and application software, need to be compiled, linked, executed and debugged together. This goes far beyond the correct individual modeling of a block using a high-level language. A successful high-level design methodology depends heavily on how well automated this framework is. SystemC is a system-level specification and design language (based on C++

classes) that has been widely adopted for high-levels of abstraction modeling. SystemC also emergence of transaction-level models (TLM) as an abstraction layer and modeling style capable of fully separating the computation part from the communication part of a model. Despite the widespread acceptance and use of SystemC and TLM, it is still very complex to create models which achieve a perfect balance between the required architectural detail for the desired types of analysis and simulation speed. This book addresses the very issues raised above. It presents a high-level design methodology, support tools and framework capable of full system modeling and design exploration, including components such as processors, custom blocks and application software. One of the main contributions of the book is the detailed description of ArchC. ArchC is an architectural description language (ADL) which allows designers to model instruction-set architectures and automatically map them into SystemC simulatable descriptions. ArchC is an ADL capable of detailed modeling of instructions and supporting processor architectures. Since ArchC generates an executable SystemC model, it can be fully integrated with other SystemC models using transaction-level interfaces, which allows the specification, modeling and simulation of complete systems. The book also explains in detail how the overall software environment works, including how the models are compiled, linked, executed together and debugged as a platform. Detailed examples using single and dual core platforms, coupled with custom accelerators and running real life applications are presented. Power modeling is also addressed in this book. Authors describe an interesting extension to SystemC called PowerSC, which allows switching and power information to be gathered and computed during and integrated with the functional simulation. Important to readers should the fact that the platform presented is Open-Source and available for download, whereas comparable systems available today are proprietary. This will certainly help researchers and developers alike to jumpstart their modeling efforts by using a readily available platform and development tools. Readers interested in a good overview of ESL methodologies as well as those interested in practical implementation details of architectural-description languages, platform modeling and support tools will be well served by this book.

Visiting Professor Reinaldo A. Bergamaschi
Odysci, Brazil/USA
University of Campinas, Campinas, Brazil

Contents

Contributors

Bruno Albertini Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil, balbertini@ic.unicamp.br

Guido Araujo Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil, guido@ic.unicamp.br

Rodolfo Azevedo Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil, rodolfo@ic.unicamp.br

Alexandro Baldassin Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil, alebal@ic.unicamp.br

Paulo Centoducatte Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil, ducatte@ic.unicamp.br

Felipe Klein Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil, klein@ic.unicamp.br

Sandro Rigo Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil, sandro@ic.unicamp.br

Luiz Santos System Design Automation Laboratory, Computer Science Department, Federal University of Santa Catarina, Florianópolis, Brazil, santos@inf.ufsc.br

Part I
System Design Representation

Chapter 1
Electronic System Level Design

Luiz Santos, Sandro Rigo, Rodolfo Azevedo, and Guido Araujo

1.1 The ESL Concept

Systems-on-chip (SoCs) became a reality in the mid-nineties, as a result of the long evolution of VLSI technology and the sheer growth of integrated circuit complexity. As it happens each time electronic design complexity impairs the expected time-to-market, the quest for higher productivity involves a combination of the following key notions:

Abstraction At design entry, the level of representation is raised to cope with the increased design complexity. As a consequence, the design flow goes through several levels and styles of representation. Hardware design representation has been raised through physical, circuit, gate, register-transfer, and functional/behavioral levels.

Reuse Pre-designed components are assembled within a new design. Reuse goes hand-in-hand with design paradigms, such as cell-based, IP-based, and platform-based.

L. Santos (✉)
System Design Automation Laboratory, Computer Science Department, Federal University of Santa Catarina, Florianópolis, Brazil
e-mail: santos@inf.ufsc.br

S. Rigo · R. Azevedo · G. Araujo
Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil

S. Rigo
e-mail: sandro@ic.unicamp.br

R. Azevedo
e-mail: rodolfo@ic.unicamp.br

G. Araujo
e-mail: guido@ic.unicamp.br

S. Rigo et al. (eds.), *Electronic System Level Design*,
DOI 10.1007/978-1-4020-9940-3_1,

Automation To overcome the error-prone and time-consuming nature of manual refinement, designers rely on Electronic Design Automation (EDA) tools. To refine the design representation from higher to lower abstraction levels, (layout, logic, register transfer, and behavioral) synthesis tools can be used. To check for equivalent functionality across successive levels of representation, a solid infrastructure of automatic verification tools is available.

Exploration The analysis of alternative design solutions with respect to area, performance and power at a given abstraction level reduces the probability that, after refinement through lower levels, the design might turn out not to meet the specified requirements, leading to redesign.

Although integrated circuit design has already witnessed abstraction level raising from the circuit level to the gate level and then to the *Register-Transfer Level* (RTL), the so-called SoC Revolution [10] required a broader paradigm shift. Since the integrated circuit became an integrated *system*, the new abstraction should harmonize the representation of both hardware and software. Abstraction offers system-level design representations, such as executable hardware-software especifications [2], transaction-level modeling (TLM) [4] and UML modeling. The *platform-based design paradigm* [14] guides reuse from the perspective of a reference system architecture. Automation asks for software-toolkit generation [9], such as retargetable compilers and binary utilities (since the software embedded in the system may be run on possibly distinct multiple target processors). Exploration addresses the joint-evaluation of hardware and software components.

This leads to the concept of *Electronic System Level* (ESL) [1], a generic term for a set of abstractions (possibly at distinct levels), which are adequate for the design representation of SoCs. It complies with the need for hardware and software co-design, while building upon legacy hardware design representation.

1.2 Requirements of an ESL Representation

A representation suitable for ESL design should provide a unique representation of the platform architecture, capture parameters handled by hardware-dependent software, allow advance performance and power evaluation on the target platform, rely on encapsulation of processor models and non-programmable components, support platform debugging, and enable virtual prototyping. Let us consider each of such aspects individually:

Uniqueness of representation Actual hardware-software co-design should rely on a unique abstraction of the hardware platform so that changes performed by either the hardware design or the software development teams could be agreed upon a same reference. It should also play the role of golden reference model for the hardware verification team.

Hardware-dependent software The representation must provide a programmer's view of the platform that not only captures the *Instruction-Set Architecture* (ISA)

of each processor, but also includes bit-accurate descriptions of I/O device registers, memory maps for all devices with configurable registers and memory layout for data, application code, and initialization routines. This allows the early development of hardware-dependent software, such as drivers and boot code, enabling actual hardware-software co-design.

Advance performance evaluation Although it is possible to obtain a rough performance estimate by running the application software on a stand-alone instruction-set simulator, a much more accurate performance evaluation is obtained when not only the processors, but all the other platform components are captured in the design representation.

Advance power evaluation In spite of the fact that power estimation is much more accurate at lower design levels, the ability to assess power earlier in the design flow helps in the identification of power bottlenecks, thereby allowing low-power design from the very beginning.

Encapsulation of processor models Since processors are the most complex system components, the expected ESL productivity gain would be seriously impaired if processor models had to be developed from scratch. Therefore, the representation should encapsulate either reused processor models extracted from a library or automatically generated processor models.

Encapsulation of non-programmable components From the perspective of hardware-dependent software development, the behavior of a component is what matters, regardless of how it will be implemented later, as far as a bit-accurate view of its I/O registers is available. Therefore, the representation should allow the functional modeling of hardware components.

Platform debugging The software to be embedded into the system could be run on the host workstation and conventional debuggers could be used to pinpoint application code bugs. Besides, code inspection could be improved by porting a debugger so that it could run on an instruction-set simulator of a target processor of a platform. Although yet useful, this conventional approach is clearly limited to uncovering software bugs. Therefore, to expose architecture design bugs, the representation should support component I/O probing or even component inner probing.

Virtual prototyping The representation should support the co-simulation of hardware and software in such a way that a virtual prototype could be used to evaluate the functionality, the performance, and the power/energy consumption of the final product. Such a prototype would contain application software, hardware-dependent software and platform architecture, elements allowing sufficiently accurate pre-evaluation without the need to dive into more time-consuming RTL-based prototyping or emulation.

1.3 ESL Design Flow

As a consequence of platform-based design, ESL flows are typically a combination of bottom-up and top-down approaches. In spite of that, Fig. 1.1 shows an idealized

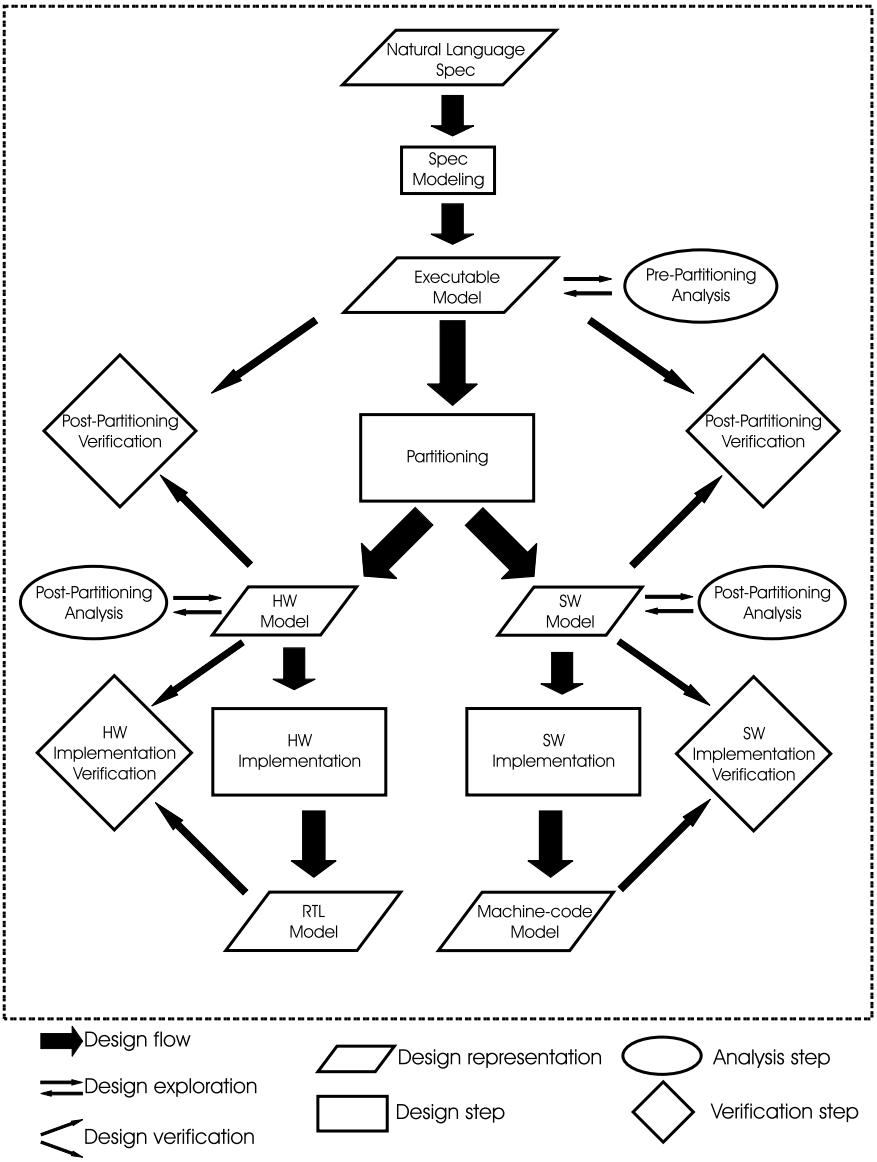

Fig. 1.1 An idealized top-down ESL design flow

top-down ESL design flow, which is used here as a frame to illustrate the main design steps rather than to advocate a particular design flow. Figure 1.1 adopts the terminology introduced in [1].

From a *natural language specification*, i.e. a document capturing the product requirements and constraints, the first design step consists in obtaining a specification

written in an executable or declarative language. (This book focuses on executable specifications, such as those obtained with SystemC [5]).

The resulting executable model, which captures the *functional specification* of a system, is submitted to a preliminary analysis step so as to provide grounds for future architectural decisions. By means of proper instrumentation, initial estimates for performance, storage space, power, and communication traffic can be obtained.

Based upon that preliminary analysis, the next design step leads to an *architectural specification* of a system, obtained from its functional specification through *hardware-software partitioning*, i.e. the mapping of algorithms to software running on target processors or to non-programmable hardware blocks. Although the resulting architecture specification captures hardware and software views, they should be built as interacting engines that execute cooperatively within a unified architectural model. To reach a threshold of simulation performance that enables hardware-software codesign, such architectural model may rely on the TLM style. (In Chap. 5, this book proposes an open-source infrastructure to build executable architectural models that comply with TLM, which is reviewed in Chap. 3).

Once an architectural specification is available after partitioning, it has to be verified with respect to the original functional specification. Essentially, such *post-partitioning verification* aims at demonstrating that the design satisfies the specified requirements.

A few building blocks are required for the architectural specification, such as processors, non-programmable hardware components, memories, and buses. On the one hand, a building block representing a non-programmable hardware component is easily modeled by its (single) behavior (e.g. the C++ implementation of its underlying algorithm). On the other hand, since processors exibit many complex behaviors, their executable models are usually generated from declarative processor models written in an *Architecture Description Language* (ADL). (This book describes an open-source ADL in Chap. 2).

The resulting architectural model also enables a *post-partitioning analysis* step, where more accurate estimates can be obtained. For instance, performance, storage space, and power can be evaluated at the light of the chosen target processors; communication traffic can be correlated with the number of transactions. This analysis leads to the exploration of alternative architectural solutions, for instance by experimenting with distinct target processors. For such an exploration, cross compilers and binary tools must be available for the target processors. (This book describes open-source infrastructure for automatically retargeting binary utilities in Chap. 6).

Post partitioning analyses of the hardware and software models provide grounds for proper *hardware and software implementation*, giving rise to an RTL model for the hardware and application binaries for the software.

The resulting implementations have to be verified with respect to the architectural specification. The ultimate goal of such *implementation verification* is to demonstrate that the implementation has no flaws. For this, the observability of a device under verification has to be increased through white-box techniques allowing the inspection of a component's inner structure. (A suitable infrastructure for white-box verification is proposed in Chap. 7).

Although, at the first glance, executable models may seem able to track behavior only (and thereby performance), they have to be extended to simultaneously track other important issues, like power consumption, for instance. (Chapter 8 shows how SystemC descriptions can be instrumented to account for power).

This book addresses a few important aspects of the ESL design flow. It focuses on languages and artifacts used as infrastructure for executable specification, hardware-software modeling, and design tools (implementation, analysis, and verification).

1.4 Target Audience, Scope and Organization

Although several ESL technologies are currently provided by EDA vendors, this book focuses on technologies available under open-source licenses, such as SystemC [5], OSCI TLM [12], ArchC [13], and PowerSC [8].

This book intends to provide grounds for further research on ESL, by means of open-source artifacts and tools, thereby stimulating the unconstrained deployment of new concepts, tools, and methodologies. It devises electronic system design from the pragmatic perspective of a SystemC-based ESL representation, by showing how to *build* and how to *use* ESL languages, models, and tools.

Other specification languages suitable for ESL design such as MATLAB [15], UML [11], SDL [7] and SystemVerilog [6] are outside the scope of this book. Proprietary ESL-compliant processor modeling, such as [3], are also beyond the intended scope.

Most of the requirements enumerated in Sect. 1.2 are covered in the next chapters, which are organized as follows:

- **ESL Specification**: SystemC is one of the most promising ESL languages. It is actually a class library that extends the C++ language in such a way that hardware and software components can be described within an executable specification. Some relevant aspects of SystemC are reviewed in Chap. 2 as a basis for the discussions in further chapters.
- **ESL-Compliant Processor Modeling**: Since describing processor models directly in SystemC would not be practical, most of processor modeling and software toolkit generation relies on ADLs. Since the ADLs designed in the mid-nineties preceded the rise of SystemC, the executable processor models generated by them were not thought to be encapsulated as SystemC modules. The ArchC language is an ADL born in the SystemC era. Instead of directly generating an executable processor model, a functional or cycle-accurate SystemC model is produced. The ArchC language is summarized in Chap. 2, while Chap. 4 and Chap. 5 explain how ArchC can be efficiently used to build ESL-compliant processor models. As an important complement, Chap. 6 shows how the ArchC ADL can be used as a starting point for the automatic generation of assemblers, linkers, and debuggers.
- **TLM-Based ESL Design**: A crucial concept to ESL is the separation between computation and communication of a system module. Since SystemC is already

built upon that concept, the definition of interfaces to encapsulate communication protocols within channels allows a convenient style of design representation: the system can be described as a set of modules (each exhibiting its concurrent behavior) that communicate by means of transactions through channels. The TLM style provides a programmer's view of the platform that allows early development of hardware-dependent software. Chapter 3 reviews the main assets of the TLM design representation.

- **ESL Design Verification**: Since the SystemC Verification Library is an already well-established verification infrastructure based on the conventional black-box approach, Chap. 7 focuses on a complementary verification aspect: the use of data introspection to enable white-box verification strategies.
- **ESL Power-Conscious Exploration**: Although SystemC can model hardware and software functionalities, thereby allowing performance-based exploration, it does not come with support for modeling power consumption. Chapter 8 shows how SystemC can be extended to capture signal transitions, to correlate them with dynamic power consumption, to add up the static power contribution from technology libraries, and to accommodate distinct power macromodels.

References

1. B. Bailey, G. Martin, A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology* (Morgan Kaufmann, San Mateo, 2007)
2. D.C. Black, J. Donovan, *SystemC: From the Ground Up* (Springer, Berlin, 2004)
3. Coware Inc., CoWare® Processor Designer. Available at http://www.coware.com. Accessed in February 2009
4. F. Ghenassia, *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems* (Springer, New York, 2006)
5. IEEE Standards Association, IEEE 1666™ Standard SystemC© Language Reference Manual. Available at http://standards.ieee.org, December 2005
6. IEEE Standards Association, IEEE 1800-2005™ Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language. Available at http://standards.ieee.org, 2005
7. International Telecommunication Union, Specification and Description Language (SDL), ITU-T Recommendation Z.100. Available at http://www.itu.int, August 2002
8. F. Klein, G. Araujo, R. Azevedo, R. Leao, L.C.V. dos Santos, An efficient framework for high-level power exploration, in *50th Midwest Symposium on Circuits and Systems (MWSCAS)* (2007), pp. 1046–1049
9. R. Leupers, P. Marwedel, *Retargetable Compiler Technology for Embedded Systems: Tools and Applications* (Kluwer Academic, Dordrecht, 2001)
10. G. Martin, H. Chang, *Winning the SoC Revolution: Experiences in Real Design* (Kluwer Academic, Dordrecht, 2003)
11. Object Management Group, OMG Unified Modeling Language™, Superstructure, V2.1.2. Available at http://www.omg.org. Accessed in February 2009
12. Open SystemC Initiative, TLM Transaction Level Modeling Library, Release 2.0 Draft 2. Available at http://www.systemc.org. Accessed in February 2009