

A VHDL-based System-Design Methodology [†]

Daniel D. Gajski

Department of Information and Computer Science
University of California, Irvine, CA, 92717

Abstract

As methodologies and tools for chip-level design mature, design effort becomes focused on increasingly higher levels of abstraction. We present a methodology and tool for system-level specification, design and refinement, based on VHDL, that results in an executable specification for each system component. The specification for each component can then be synthesized into hardware or compiled to software. We highlight advantages of the proposed methodology compared to current practice.

1 Introduction

The focus of design effort on higher levels of abstraction has led to the need for a system-level methodology and supporting tools. There are two main steps in system-level design. The first is *functionality specification*, which is the task of describing the desired system behavior in some form. The second is *system design*, which is the task of implementing this functionality with system components such that design constraints are satisfied. Example system components include standard processors and microcontrollers, memories, buses, and custom ASICs. The domain of these two steps is shown in Figure 1. The result of system design is a set of system components, each with its own functional specification. Implementation of each component follows. A standard component requires software compilation of the functional specification into machine code, whereas custom components require synthesis of the specification into register-transfer structure. The first task is accomplished with standard compilers while the second one uses high-level and logic synthesis.

There are two very different system-level design approaches in current practice. In one approach, the system's functionality is first implemented with interconnected register-transfer or gate-level objects, and

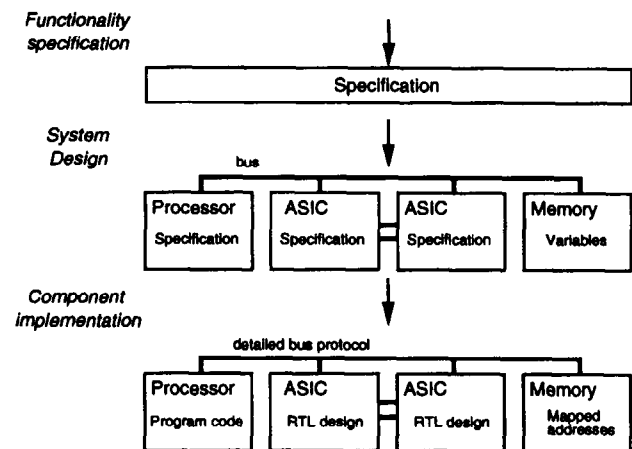


Figure 1: System-level domain

this structure is then partitioned among system components. However, once structure is obtained only minor changes can be made to the system's performance through introduction of redundant objects or through repartitioning. More substantial changes require knowledge of high-level functional and timing information, but such information can not be discerned from the structure. A second drawback to this approach is it doesn't consider software implementations.

In the other approach, the system's functionality is first partitioned among system components, and each component is then implemented as structure or as software, depending on the component type. While overcoming the drawbacks of the structure-first approach, current practice of this approach involves mostly informal and manual techniques. The functionality is informally specified using a natural language such as English, and system design is done manually using mental or hand-calculated estimations for quality metrics such as performance, size, and power. Drawbacks of such techniques include the lack of early functional verification, the lack of good feedback with regards to quality metrics that result from design decisions, the lack of automated tools to reduce design time, and the lack of good documentation of functionality and design decisions to aid in concurrent design and in re-

[†]This work was supported by the SRC (grant #91-DJ-146), NSF (grant #MIP 8922851-01) and California Micro (grant #91-040, #91-041).

design.

Several research efforts have focused on overcoming one or more of these drawbacks. Simulation environments have been developed to encourage early system simulation of hardware and software components for functional verification [1, 2]. An architectural template and tools environment for rapid prototyping have also been suggested [3]. Functional partitioning approaches have been introduced for multiple custom chips [4, 5], and for multiple processors [6]. Issues for functional partitioning among hardware and software components have been discussed [7], and prototype partitioning systems have been developed [5, 8]. Frameworks have been proposed to support the process of controlling and interfacing various system-design tools [9]. An overview of this work can be found in [10].

The methodology and tool we present can be used in conjunction with the simulation, prototyping, and framework environments described above. Our work differs from other previous efforts in several key points.

First, we handle exploration of various implementations of the *three* aspects of functionality, namely behavior, data, and communication, rather than focusing on behaviors alone as in most previous work.

Second, our technique is applicable to a variety of system-component technologies, not just a fixed set of hardware or software components of one technology. The above two points provide a seamless exploration of system-design options, which includes hardware/software codesign.

Third, the end result of our approach is a *refined specification* in which interconnected system-components are functionally specified, permitting further verification and encouraging concurrent design.

Fourth, we use a new model (PSM) for specification that can describe both hardware and software functionality, at varying levels of abstraction, in a uniform manner. This model differs from previous ones which were well suited for either software or hardware but not both.

In this paper, we present a new system-level methodology. It is highlighted in Figure 2, where the boxed items represent a replacement of informal and manual techniques with well-defined and automatable ones. We define and describe the major system-design tasks, as well as a suggested ordering of those tasks. We describe an environment to support the methodology.

2 System Design

We use an executable specification rather than a natural language to specify system functionality. An executable specification is one which captures the

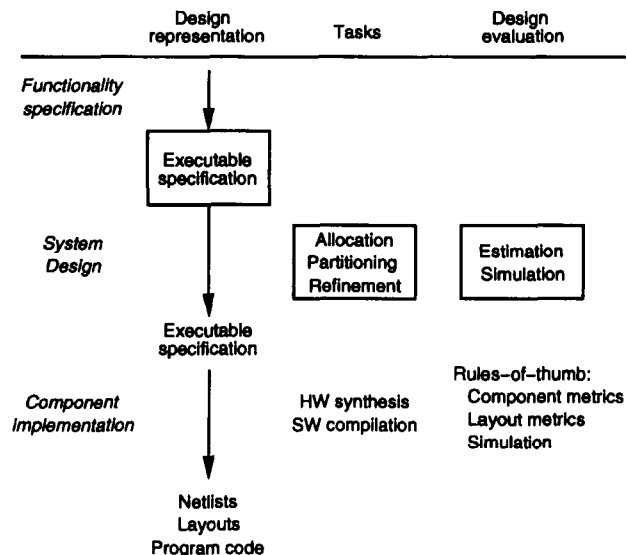


Figure 2: Proposed system-level methodology

functionality of the system in a machine-readable and simulatable form. It has several advantages over a natural language. First, simulation enables early verification of the correctness of the system's intended functionality. Second, the specification may serve as an input to synthesis tools, resulting in significantly reduced design times. Third, the specification serves as documentation by providing a precise description of intended functionality.

Since no language currently supports all the embedded system characteristics, we developed a VHDL front-end language called SpecCharts [11].

Other languages may be used to capture the same functionality, however, SpecCharts is in closest accord with our system-design methodology, and yields the most concise and readable specifications.

| Functional objects | System-design tasks | | |
|--------------------|---------------------|-------------------------|-----------------------|
| | Allocation | Partitioning | Refinement |
| Variables | Memories | Variables to memories | Address assignment |
| Behaviors | Processors | Behaviors to processors | Interfacing |
| Channels | Buses | Channels to buses | Arbitration/protocols |

Figure 3: System-design tasks

System design is the task of mapping the functionality, as captured in an executable specification, to some set of system components such that design constraints on parameters such as monetary cost, performance, and power are satisfied. Our approach to system design consists of three well-defined tasks on three classes of functional objects, as summarized in Figure 3. The three classes of functional objects that comprise any executable specification are **variables**,

behaviors, and **channels**. Variables store data, behaviors transform data, and channels transfer data between behaviors. In our terminology, a behavior is a non-trivial algorithmic-level computation that together with other behaviors describe all system actions (identical to the “task” concept described in [7]). It corresponds to a block of statements in the specification such as a loop body, procedure, or process. For each of these objects there are three tasks to be performed: allocation, partitioning, and refinement.

Allocation adds system components to the design. One class of system component consists of memories, such as RAMs, ROMs, register-files, and registers. Memories are used to store scalar and array variables. Another class of component consists of standard processors and microcontrollers as well as custom ASIC “processors”. These standard/custom processors are used to implement behaviors. A third class of “component” consists of physical buses. Buses are used to implement communication channels.

Partitioning maps each class of functional objects to allocated components. Variables are mapped to memories, behaviors are mapped to standard/custom processors, and channels are mapped to buses. Each mapping is many to one. Standard partitioning algorithms, such as clustering or simulated annealing, can be applied. Clustering may use any of various closeness criteria [12]. For behaviors, common criteria include interconnection, communication, sequentiality, and hardware sharability. For variables and for channels, common criteria include sequential access, common accessors, and width similarity.

Refinement adds new behaviors to maintain correct functionality for a given allocation and partitioning. Variables partitioned among memories require memory address translation. Behaviors separated among components must be modified to maintain correct communication. Channels mapped to buses require interface synthesis to determine communication protocols, and arbiter synthesis to resolve any simultaneous bus requests. A refined specification is then generated consisting of a set of interconnected system-components, each functionally specified.

There is no fixed ordering of the system-design tasks. One ordering which we have found leads to good results is shown in Figure 4. After the functionality is specified, large variables are mapped to memories such that variables which satisfy closeness criteria are mapped to the same memory. Channels are mapped to buses in a similar manner. Then processor or ASIC components are allocated and behaviors are partitioned among those components. Variable or channel repartitioning may follow in order to further improve the design. Interface and arbiter synthesis are

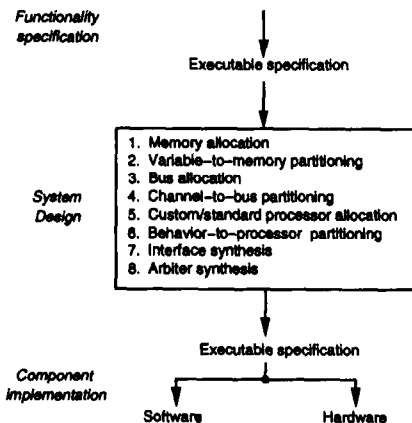


Figure 4: One possible ordering of tasks

performed to complete the functional specification of each component.

Throughout the entire process, allocation and partitioning tasks are guided by estimations of design quality metrics such as area, performance, pins, and power. Accurate yet fast estimation techniques are a subject of intense research; some results for both hardware and software implementations are described in [10, 13]. The estimations are incorporated into an objective function used for design evaluation. A common objective function is one which favors designs with the smallest amount of constraint violations.

To further evaluate design decisions, a refined specification can be generated and simulated between any tasks during the design process.

The resulting system components can be implemented with either automated or manual techniques. Since the components are defined as an executable specification, synthesis or compilation follow very naturally.

3 Conclusion

A set of tools to support our methodology has been implemented as the SpecSyn system-design environment. It consists of partitioners which support several algorithms including clustering, group migration, and simulated annealing; estimators for the quality metrics of execution-time in hardware or software, hardware area, software instruction and data memory size, and pins; and prototype tools to synthesize arbiters and interfaces. The partitioners, estimators, and interface/arbiter tools are implemented in C with 16000, 19000, and 8000 lines of code, respectively. Routines for internal representation of the specification require 30,000 lines of code. SpecSyn has been released to several companies.

We have applied the methodology with the SpecSyn environment on several examples, including a medi-

cal instrument for measuring bladder volume, a fuzzy-logic controller, a RISC signal processor, an interactive TV system, a microwave-transmitter controller, and an answering machine. Design quality is comparable with manual designs and design-time is up to an order of magnitude less. Numerous manual allocations and partitionings and hundreds of automatically generated ones can be evaluated in just minutes.

There are several advantages to using a system-level methodology that refines an executable specification through the system-design tasks of allocation, partitioning, and refinement. First, early functional verification through simulation is possible, which minimizes the occurrence of time-consuming functional changes later in the design process. Second, the resulting system-component executable specifications are precise so they enable concurrent design and minimize integration problems. Third, the specification is machine-readable so automation tools can be used to reduce overall design time and estimators used to rapidly explore many more possible designs. Fourth, the well-defined system tasks combined with the executable specification lend themselves to excellent documentation of system-design decisions and intended functionality, which is especially important for re-design.

We envision a conceptualization environment that allows designers to quickly explore and evaluate potential designs. This requires work on three parts of such an environment: (1) a component base with VHDL models for different technologies, (2) fast estimators of quality metrics for different architectural styles and technologies (e.g. custom layout, gate-arrays, FPGA's, and standard components), and (3) an appropriate human interface for display of quality metrics and different user views. The SpecSyn environment is a first step in this direction.

4 Acknowledgement

I would like to acknowledge the contributions of Frank Vahid, Sanjiv Narayan and Jie Gong, who discussed the main concepts and who implemented the SpecCharts language and the SpecSyn environment and performed all of the experiments. Without their efforts, this paper would not have been possible.

I would also like to acknowledge the continuous support of the SRC and NSF for this project. I am grateful to Peter Verhofstadt and Bob Grafton for their vision and support of system level methodology.

References

- [1] A. Kalavade and E. Lee, "A hardware/software code-sign methodology for dsp applications," in *IEEE Design & Test*, 1993.
- [2] R. Gupta, C. Coelho, and G. DeMicheli, "Synthesis and simulation of digital systems containing interacting hardware and software components," in *DAC*, pp. 225-230, 1992.
- [3] M. Srivastava and R. Brodersen, "Rapid-prototyping of hardware and software in a unified framework," in *ICCAD*, pp. 152-155, 1992.
- [4] E. Lagnese and D. Thomas, "Architectural partitioning for system level synthesis of integrated circuits," *IEEE Trans. on CAD*, July 1991.
- [5] R. Gupta and G. DeMicheli, "Hardware-software cosynthesis for digital systems," in *IEEE Design & Test*, pp. 29-41, October 1993.
- [6] S. Prakash and A. Parker, "Synthesis of application-specific multiprocessor architectures," in *DAC*, pp. 8-13, 1991.
- [7] D. Thomas, J. Adams, and H. Schmit, "A model and methodology for hardware/software codesign," in *IEEE Design & Test*, pp. 6-15, 1993.
- [8] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," in *IEEE Design & Test*, pp. 64-75, December 1994.
- [9] M. Jacome and S. Director, "Design process management for cad frameworks," in *DAC*, pp. 500-505, 1992.
- [10] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*. Prentice Hall, 1994.
- [11] S. Narayan, F. Vahid, and D. Gajski, "System specification with the speccharts language," in *IEEE Design & Test*, Dec. 1992.
- [12] D. Gajski, J. Gong, F. Vahid, and S. Narayan, "The specsyn design process and human interface." UC Irvine TR 93-3, 1993.
- [13] W. Ye, R. Ernst, T. Benner, and J. Henkel, "Fast timing analysis for hardware-software co-synthesis," in *ICCD*, pp. 452-457, 1993.