Embedded Systems Education for the Future

WAYNE WOLF, FELLOW, IEEE, AND JAN MADSEN

This paper describes lessons we have learned over the past several years about teaching the design of modern embedded computing systems. An embedded computing system uses microprocessors to implement parts of the functionality of non-general-purpose computers. Early microprocessor-based design courses, based on simple microprocessors, emphasized input and output (I/O). Modern high-performance embedded processors are capable of a great deal of computation in addition to I/O tasks. Taking advantage of this capability requires a knowledge of fundamental concepts in the analysis and design of concurrent computing systems. We believe that next-generation courses in embedded computing should move away from the discussion of components and toward the discussion of analysis and design of systems.

Keywords*—*Computer engineering education, embedded computing, microprocessors.

I. Introduction

Education about embedded computing is ripe for a change. Microprocessors that were once prized centerpieces of desktop computers are now being used in automobiles, televisions, and telephones. This huge increase in computational power can be harnessed only by applying structured design methodologies to the design of embedded computing systems. Over the past few years, we have experimented with new ways to teach embedded computing that embrace this change in the digital design landscape. This paper summarizes the lessons we have learned.

In the late 1970's, Mead and Conway espoused the need for tall, thin very large scale integration (VLSI) designers—these people would have some knowledge at every level of abstraction in the design process and so could see how to optimize the system globally. Today, we need to train tall, thin embedded systems designers. As with VLSI system design, embedded systems designers must have knowledge of the complete design process so that they can make global rather than local decisions. The biggest difference between VLSI design circa 1980 and embedded system design circa 2000 is the level of complexity of components: VLSI design in 1980 concentrated on transistors, through gates, up through chips with tens of thousands of transistors; today, those would be components of still larger systems with millions or tens of millions of transistors and thousands or hundreds of thousands of lines of software.

Section II will survey the state of the art in embedded computing systems, covering both architectures and design methodologies. Section III briefly summarizes our history of embedded computing courses, which serves as background for our discussion in Section IV of topics and principles for teaching embedded system design. Section V provides some additional observations unique to teaching graduate or other advanced courses in embedded systems and codesign.

II. Embedded Computing Systems

Microprocessors are everywhere—*The New York Times* estimated several years ago that the average American came into contact with 100 microprocessors per day, and that number grows rapidly with time. Microprocessors have been widely used for over 25 years; consumers have used microprocessor-enhanced appliances for quite some time. However, the role of embedded microprocessors has fundamentally changed as advances in VLSI technology have allowed us to produce high-performance microprocessors cheaply. Early microprocessors were used for controlling analog subsystems; today, much more of the system's work is done in the digital domain.

Early microprocessor applications emphasized input and output (I/O), not computation. A typical microcontroller would be used to interface to several I/O devices and to perform basic sensing operations. Relatively simple transformations would be made on the data by the microprocessor. Most of the work was done by external analog devices; the job of the microprocessor was to control and sequence those external devices to provide a higher level of overall functionality. Typical microprocessor functions might include reading front-panel buttons and setting front panel LED's, controlling stepper motors, and controlling relays.

Microprocessor courses started in the 1970's soon after the appearance of microprocessors. Many of these courses

Manuscript received April 7, 1999; revised July 2, 1999. This work was supported by an Educational Development Grant from the National Science Foundation and a Large Instrumentation Grant from Hewlett-Packard.

W. Wolf is with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08544-5263 USA.

J. Madsen is with the Department of Information Technology, Technical University of Denmark, Lyngby, Denmark.

Publisher Item Identifier S 0018-9219(00)00203-6.

0018–9219/00$10.00 © 2000 IEEE

emphasized hands-on experience with a particular microprocessor; these courses would discuss programming and hardware interfacing for the microprocessor used in the laboratory. A great deal of emphasis was also placed on the characteristics of I/O devices. Discussion of I/O devices generally emphasized their mechanical and analog electronic properties even more than their digital behavior.

Peatman's well-known textbook from 1977 [4] gives us one good window into microprocessor-based systems education in that period. Peatman's book is not overly tied to a particular microprocessor and it does mention some of today's hot topics, such as high-level language programming and deadline-driven computing. However, the book puts a great deal of emphasis on the I/O devices themselves, including their electromechanical properties and interfacing these devices to the microprocessor. The discussion of programming emphasized low-level topics such as arithmetic and table construction and use; one of the most interesting sections describes how to use a software state machine to parse keystrokes from a user interface.

Emphasizing details of a particular microprocessor fits with the limitations of early microprocessors. These microprocessors operated on small words. They had limited address spaces, corresponding to the expense of memory. They ran at relatively slow clock rates. The limitations of the architectures made it difficult to compile efficient code, so most programs were written in assembly language. All these limitations conspired to require considerable handcrafting of microprocessor-based systems. Given that design had to jump quickly into the fine points of the microprocessor being used in order to make any progress, it was natural for microprocessor-based system design courses to emphasize the particulars of one microprocessor over the common characteristics of microprocessor-based systems.

Now, however, the scene has changed. There are probably more embedded microprocessor architectures today than ever, but this only allows us to see commonality in approaches. Higher levels of integration provide larger word widths, larger address spaces, and larger memories. Clock rates have increased to allow complex programs to be run on the time scales allowed by embedded computing systems. As VLSI implementations and compiler technology have improved, high-level language programming has become more popular, even for 8-bit microcontrollers [1]. All these changes allow us to study embedded computing as a systems discipline in which components can be characterized and assembled to meet design goals.

High-performance RISC CPU's and digital signal processors (DSP's) are now commonly used in embedded systems. Of course, there will always be a place for small microprocessors. However, they will in many cases be used as components in larger systems. There are two major reasons to split an application's computation across multiple CPU's even when one CPU is available that could meet all performance deadlines: the application may have physically distributed I/O (consider an automobile, for example, in which microprocessors are used all over the car), in which case either the expense of wiring or the delay of transmitting values back and forth demands local processing of data; or because it may be cheaper to use several smaller CPU's, each of which runs one or a few nonconflicting jobs, than it is to use one large CPU that runs several contending programs.

Several examples show the complexity and varieties of system architectures that have been developed for embedded computing. DSP's or other CPU's are often used to perform on-board signal processing. This is true both for telephones, in which DSP's perform filtering and signal generation functions, and for printers, which use processors for smoothing and image enhancement.

An article in the *Hewlett-Packard Journal* [6] describes the hardware and software design of a large ink-jet plotter. The plotter used a 32-bit RISC processor to perform noncontrol oriented tasks such as parsing the page description language, plus two custom ASIC's and two microcontrollers to perform special-purpose and real-time control applications. In such a complex system, the division of tasks among processing elements is crucial—bad decisions can lead to overly expensive hardware and also to unacceptably high bug rates. The designers had to create a custom debugging environment to be able to debug as much software as possible before the hardware platform was complete.

Internet appliances—smart devices which are accessible over the Internet—will become an increasingly important category of embedded system. An article on an Internet-accessible video camera [19] describes the design of such appliances. The camera is a stock unit which connects to a PC parallel port. A PC-style hardware platform was used, with a 486-class CPU as the compute engine. The software is implemented in Java; the system requires about 1.5 Mbytes of memory. The Java interface allows other Internet nodes to control the camera and the camera to generate an HTML page showing the current frame.

As components become more complex, methodology becomes increasingly important. Small microcontrollers permit (and may even demand) a handcrafted approach to design. When high-performance CPU's runs tens of thousands of lines of code, a more structured methodology becomes a necessity. While many methodological variations have been proposed, embedded system methodologies increasingly follow the models of VLSI and software design, using top-down architectural refinement combined with bottom-up cost information. Fig. 1 illustrates a basic methodology in which major architectural decisions are made by jointly considering hardware and software components, and the system is verified as a whole using cosimulation and coverification techniques. Hardware and software components can be implemented separately, and embedded systems are often designed to allow product improvements through software upgrades, but it is important to understand how architectural choices for hardware and software affect each other.

Thoma and Fuchs [3] of BMW describe the characteristics of modern automotive electronics systems and describe a new methodology for the design of these systems. They state that modern upper class automobiles use up to 70 electronic

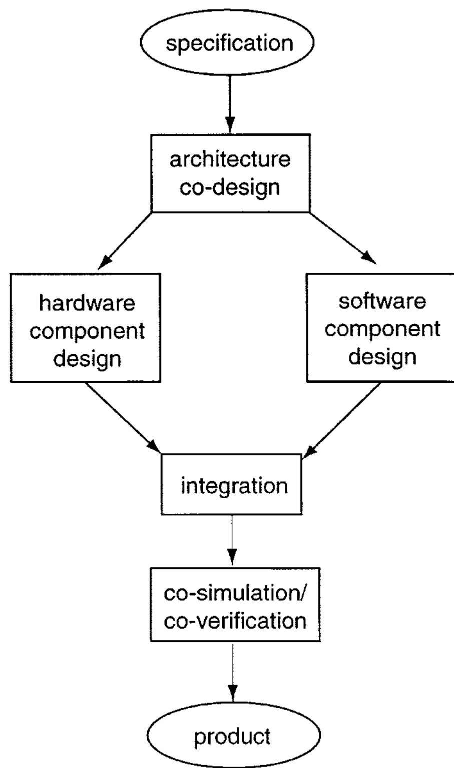

Fig. 1. A methodology for embedded system design.

control units (ECU's) communicating over several busses; 30% of the added value in the automobile is in the electronics. They propose a design methodology in which the development of the network of ECU's is separated from the design of the software running on that network. The software speaks to the ECU platform using an application programming interface (API). Separation of hardware and software development allows the designers to move software from one ECU to another to improve ECU utilization globally; it also allows manufacturers to add value to generic components by adding software that implements unique features. They also emphasize the use of abstract specifications for the system functionality, using tools such as Matlab to describe algorithms.

Methodological issues will become even more important with the advent of systems-on-silicon. A system-on-silicon integrates memory and logic, perhaps with analog interfaces, to create a complete system on a single chip. These systems will contain hundreds of millions of transistors; the *National Technology Roadmap* predicts that chips with 1.4 billion transistors will be on sale in 2012 [7]. Especially since these chips will have to be designed in very limited amounts of time, making extensive use of embedded CPU's (as well as other sorts of intellectual property) will be critical to maintaining adequate designer productivity. Clearly, shifting large blocks of a design from a gate-oriented methodology to a software-oriented methodology will require a major shift in not only how systems-on-silicon designers work, but also in how they are trained.

III. Our Experience

A. *Experience at Princeton University*

Wolf has taught embedded systems design courses to undergraduates at Princeton over the past several years, starting in the early 1990's. This experience helped to shape some of the major themes of an embedded computing course; they also showed the dangers of trying to cram too much material into a single semester. Students in the course had no previous exposure to computer architecture, so the course had to introduce the basics of instruction set processors as well as concepts unique to embedded systems. The course covered a broad range of topics.

- Early lectures surveyed the basic principles of microprocessors, including instruction sets, busses, interrupts, assemblers, and the rudiments of assembly language programming. One or two labs were used to introduce the basics of microprocessor hardware and software.
- Another set of lectures covered the aspects of high-level language programming related to embedded systems design. This included a survey of compilation techniques so that students had some appreciation of how C programs mapped into instructions. Some labs allowed them to practice writing programs for the microprocessor. One particularly interesting lab allowed students to study how the behavior of the microprocessor cache affected performance: by changing a simple loop, they could change the cache hit rate and see the resulting change in performance on the bus with a logic analyzer.
- We talked briefly about systems architecture. On the hardware side, we considered how CPU's, devices, and memory cooperated to perform useful tasks. On the software side, we concentrated on how to decide what should go into an interrupt handler versus what happened in a background task.
- We used the final few weeks of the course to introduce some topics normally covered in software engineering classes: testing; safety; and specification.
- A capstone lab required student teams to design a small project. Sample projects included a digital telephone answering machine and a multiple elevator controller.

We used two different hardware platforms over the years: a Motorola 68020 programmed in assembly language and an Intel i960 programmed in C. We found C programming to allow us to get much more deeply into system design without worrying so much about instruction sets.

Wolf and Wolfe also used a graduate course to create a significant embedded system. Over the course of the semester, we designed and mostly implemented a PC-based personal branch exchange (PBX) [22], including the design of a printed circuit board for a telephone line interface and real-time software for switching. Some remaining bugs in the circuit board prevented final testing, but we walked through a great deal of the design process and learned quite a

bit. Such hero experiments require time and energy, but they are rewarding when they are carried through with sufficient vigor.

B. Experience at Danish Technical University (DTU)

In the summer of 1998, Madsen and Wolf taught a three-week course at DTU on embedded systems and hardware/software codesign. The course was targeted to master's-level students, who were assumed to know the basics of computer architecture, logic design, etc. However, we tried to cater to a fairly broad spectrum of students at that level of experience: we wanted to cover the basics of embedded system design at a level suitable to those who had no significant experience in building microprocessor-based systems.

Since students took only one course during this period, we could cover quite a bit of material. The major goal of the course was completion of a project selected by the students; several labs helped them build skills that they would need to complete the project. We used the Analog Devices Sharc EZ-Kit Lite board for all projects, since it has a good C compiler and is inexpensive. We had about 25 h of lectures over the three week period. Each week had a different emphasis.

- The first week concentrated on the fundamentals of the microprocessor instruction set, interrupts, embedded programming, etc. We spent some time talking about program performance—we considered the execution time of various instructions as well as the effects of caching and interrupts on performance. Students were familiar with computer architecture in general, but none of them had experience with the Sharc and few had significant hands-on experience with embedded microprocessors.
- The second week was a bridge week in which lectures looked more at system design while labs built skills. Lectures here emphasized architectural and software skills, such as how to determine what should go into an interrupt routine. We introduced basic real-time systems theory as a bridge to real-time multitasking embedded computing. The labs built up to the implementation of a very simple software modem.
- The third week had the fewest lectures, which concentrated on presentations of research in hardware/software codesign; the students spent much of the week designing, building, and debugging their projects. None of the projects required hardware construction-all used the on-board A/D facilities and serial port provided by the development board. This allowed the students to concentrate on the creation of complex functionality through programming. Projects included a speech compressor, an audio encoder, etc.

IV. Teaching Embedded Computing System Design

A. Basic Principles

Just as computer technology changes, so also does computer engineering education. Today's computer architecture courses spend more time on instruction set design and pipelining and less time on topics like I/O. As a result, before getting to the significant examples in embedded computing, a course must first cover some basic principles and techniques that have not been learned in other courses. For older teachers, this may require some adjustment in expectations, since today's students have much less hands-on experience with raw CPU's than was available to students before PC's became ubiquitous. Not only microprocessor design courses, but also minicomputers provided a programming environment in which they could program free from operating system restrictions and with full access to the halt button. Today, students must rely on an embedded systems course for such experience-they usually do not enter the course with that knowledge.

We believe that major principles to be conveyed in an introductory embedded systems course include:

- *I/O:* Modern computer architecture courses do not place a great deal of emphasis on I/O; they tend to cover basics such as disk performance and the effect of I/O on the CPU pipeline. Students need to understand the functionality of interrupts, the properties of device drivers, and the performance tradeoffs of interrupt-driven and polled I/O.
- *Concurrency:* This is closely related to I/O, since a great deal of concurrency is caused by input and output. However, concurrency is not limited to I/O; in fact, it is the combination of computation and I/O that makes modern embedded systems challenging and differentiates them from microcontroller-based systems. Learning how to get computations done despite the continued presence of I/O interrupts is an important skill. Students need to understand how concurrency is created by hardware and software, how it affects overall system performance, and how it complicates debugging.
- *Deadline-Driven Concurrency:* This, too, is related to I/O, since most deadlines have something to do with the requirements of input and output. Writing programs to meet a performance deadline is very different from the non-real-time programming that most students are used to. Students need to understand how architectural features such as pipelines and caches affect program performance; how to analyze a program to estimate its execution time; and how to engineer a program to meet its performance requirements.

We believe that the best way to convey these principles is to use more lectures and homework and somewhat less lab work than is typical of component-oriented microprocessor curricula. Rather than concentrating on hands-on experience with a particular microprocessor, we believe that some basic principles can be used to provide a framework, followed by reinforcement with homework and labs. For example, the fundamentals of real-time scheduling, such as rate-monotonic scheduling [21], both explain features of real-time operating systems and provide students with an appreciation of the theory underlying embedded system design. Fig. 2

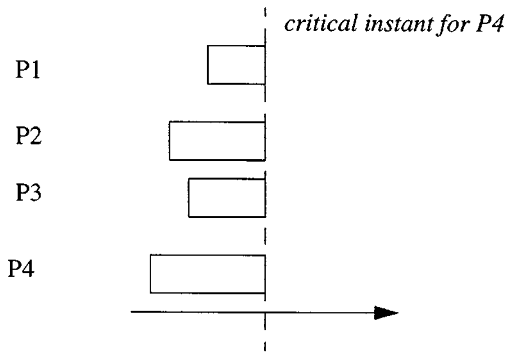

Fig. 2. Critical-instant analysis for rate-monotonic scheduling.

illustrates critical-instant analysis, the fundamental concept underlying the proof of rate-monotonic scheduling's optimality. $P1$–$P4$ are a set of processes running on a single CPU; the worst-case response time for $P4$ happens when all higher priority processes are activated at the same time as $P4$, since they must all run to completion before $P4$ can begin. This concept can be used to prove the properties of rate-monotonic scheduling: that it is optimal under the set of assumptions made; and that a certain percentage of the CPU goes idle waiting for processes to be activated. Such techniques (and other topics, such as software testing, described below) provide the opportunity to use equations and algorithms as an underpinning for embedded system design. Discussion of basic principles can be leavened with discussion of embedded system examples such as those of the last section, followed by hands-on experience in the lab. These principles not only provide students with some basic techniques with which to understand particular design situations, they also make it easier to teach embedded computing by reducing the dependency on complex laboratory hardware setups. The IEEE workshops on real-time systems education provide a good source of material on the teaching of these subjects.

It is quite possible and advantageous to avoid the discussions of the mechanical aspects of I/O devices that permeated early embedded systems courses. Today, most embedded system designers get relatively high-level specifications for the devices in their system; this will be increasingly true in the systems-on-silicon era. Emphasizing digital techniques also helps make the course more appealing to software-oriented students who may feel uncomfortable with unfamiliar mechanics. Emphasizing a digital view of the world may make the course somewhat less successful as a service course for mechanical engineers, physicists, and others who want to use microprocessors for interfacing, but this is probably inevitable. Many electrical engineering (EE) and computer science (CS) students of the near future will become designers of high-performance embedded systems and systems on silicon, an area that is rapidly becoming a field unto itself.

B. Additional Concepts

There are some other major ideas that are not as essential to success in an introductory embedded systems course but are nonetheless important. Both software testing and system specification techniques are important to the practice of embedded system design, but the amount of coverage they receive in an introductory course depends on the background of the entering students and the amount of time available. Some of the topics echo material in software engineering courses, but there are reasons to at least consider talking about them separately in an embedded systems course. First, software engineering typically places less emphasis on performance and real-time computing. Second, even if students do not become masters of these topics in the embedded systems course, some exposure will teach them basic concepts that will help them learn more about the topics later.

Software testing is probably the most important of these secondary topics and deserves at least some place in an introductory embedded systems course. Given that most embedded systems are programmed in ROM, freezing bugs in place, it is important for students to learn something about testing the embedded systems they design. In the preface to their widely used book on software testing [5], Kaner *et al.* state that they "have yet to meet a computer science graduate who learned anything useful about testing at a university." They go on to criticize *Computing Curricula 1991*, a curriculum formulated by the IEEE Computer Society and ACM, for its scanty coverage of testing techniques. We agree that testing is an important topic and that embedded system designers in particular should learn some fundamental testing techniques. The Kaner *et al.* book covers topics that are specific to PC testing such as testing printers; it also spends a great deal of time on methodologies.

Our approach to teaching testing for embedded systems has been to spend one or two lectures on the subject to cover fundamentals, then to have students practice these skills in both homework and labs. While we mention black-box techniques like random testing and regression testing, most of the discussion emphasizes clear-box techniques that analyze program structure to extract relevant tests. Fig. 3 shows control flow and data flow graphs for a sample program. Various techniques exist for analyzing these graphs to ensure that the graph is covered by some metric. For example, a common control flow coverage goal is ensuring that each block is executed at least once; data flow coverage requirement typically requires not only that each operator be executed at least once, but that arithmetic operators be tested with various combinations of positive and negative arguments. By relating software testing to the analysis of high-level languages for compilation and performance analysis, we emphasize the relationships between front-end design and back-end verification; basing the techniques on previously covered material also makes them easier for students to absorb. We have observed that students who have not been taught about testing are almost always very poor testers. While they understand intellectually the large number of states that a program can assume while executing, they invariably come up with a very small number of simple tests that do not do justice to the testing problem. Teaching students a few basic techniques for software test coverage substantially improves their ability to uncover bugs. Other well-known books on software testing include [8] and [9]. Software safety is an important topic

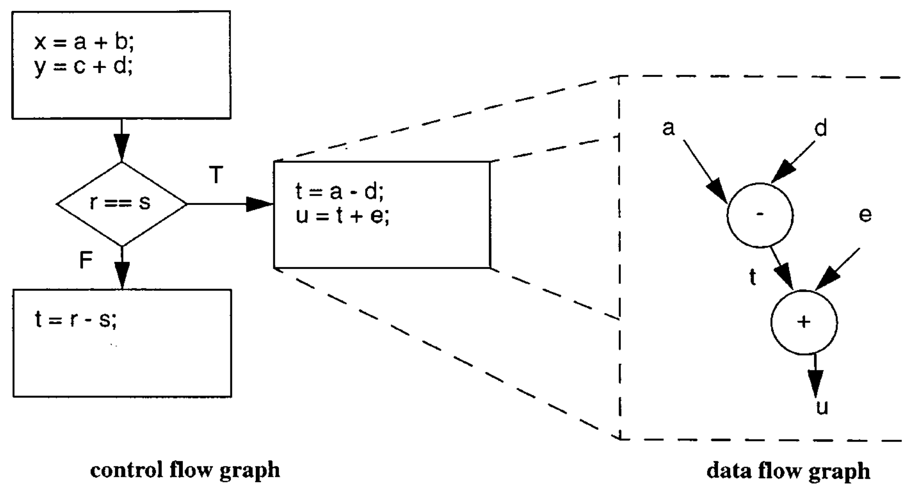

Fig. 3. Control flow and data flow in a program.

that is related to testing; testing is necessary for safety but certainly not sufficient. Articles by Leveson *et al.* [10], [11] make a good way to introduce students to the challenges inherent in building safety-critical computing systems.

In our experience, design specification techniques are tougher to teach in an introductory course. Students are usually not familiar with the concept of specifying software; they typically need experience with the design of a large body of software before they see the need for specifications. However, some mention of architectural issues gives students some framework with which to understand problems in the architectural design of embedded computing systems. A study of design specification techniques teaches several things: how to extract structure from an incomplete functional description of a device; how to relate that functional information to other requirements like deadlines; and how to take advantage of that structure when creating the architecture for the system. There are several books and papers that might be referenced in these discussions. Davis's book [13] gives a good introduction to the requirements process, which precedes specification and is intended to capture the basic capabilities of the system. Hatley and Pirbhai [15] described an early methodology for the design of real-time systems. Booch [12] developed a pioneering methodology based on object-oriented techniques; Rumbaugh *et al.* [17] created another important methodology for object-oriented design. Selic *et al.* [18] created a methodology which applied object-oriented techniques to the design of real-time systems. The Statechart, introduced by Harel [14], is a well-known technique for specifying control. Leveson *et al.* [16] developed a design methodology inspired by Statecharts; they used this methodology to capture and verify the design of the TCAS-2 collision avoidance system that has been adopted by the Federal Aviation Administration. The Unified Modeling Language (UML) [25] is a new object-oriented modeling language that promises to become an important specification and design technique.

C. Laboratory Experience

Our early attempts at teaching embedded computing included some laboratories that involved hardware construction, such as interfacing a device to the microprocessor bus. However, experience suggests that hardware construction can get in the way of system design concepts. It is clearly important for students to understand how embedded hardware works. However, much can be learned by observing prebuilt hardware. One of the most valuable labs in the Princeton courses required students to use a logic analyzer to observe cache behavior: by changing the number of instructions in a loop, students could see varying cache miss rates through accesses to off-chip memory. Using prebuilt boards allows students to concentrate on building complex programs that exhibit interesting concurrency. As technology improves, it is becoming increasingly hard to build interesting hardware at all. Even in the absence of systems on silicon, integrated circuits are increasingly provided in surface-mount packages that are difficult to solder by hand.

V. Teaching Advanced Embedded Computing

Graduate-level courses in embedded systems naturally keep closer to the cutting edge of research. In embedded computing, this has two aspects. First, more complex hardware and software architectures can be discussed. Many modern embedded systems are multiprocessors, including both CPU's and ASIC's, and take advantage of real-time operating systems. Second, computer-aided design (CAD) tools will play an larger role in embedded system design. Hardware/software co-design is both a methodology and a collection of CAD algorithms that are being developed now to meet the challenges of future embedded systems design.

Systems on silicon will allow much higher performance and advanced embedded systems architectures to be built. The availability of multiple processing elements and large amounts of memory on a single chip will make it possible to

create sophisticated multiprocessors for embedded applications. However, embedded computing platforms are typically heterogeneous multiprocessors, not the regular architectures found in high-performance computing. Cost reasons dictate that the selection of processing elements, memory system, and the interconnections between them should all be specialized for the target application. The specialization of the architecture to the task makes designing a system challenging and developing CAD techniques for these systems even more challenging.

Systems on silicon are also the foremost impetus in the development of CAD techniques for embedded systems. As architectures become more complex, the difficulty of designing embedded systems increases, but the cost of errors skyrockets when the system is integrated on a single chip. While design errors can be fixed in minutes on a printed circuit board, design turns take weeks or months in silicon and at much higher cost. The solutions are co-simulation (simulating hardware and software components together as a single system), co-synthesis (designing hardware and software architectures simultaneously to meet functional and nonfunctional requirements), and co-verification (verifying the characteristics of a combined hardware/software system).

Even without a research-oriented seminar course, it is possible to convey the fundamentals of systems on silicon and codesign to students with good undergraduate preparation. Analysis techniques can be used as a stepping stone to understanding both synthesis and verification of hardware/software systems. Graduate courses can also delve more into concepts such as specification given the greater experience and maturity of the students.

A special issue of this PROCEEDINGS [23] covered the state of the art in hardware/software co-design. Other sources include proceedings of the CODES/CASHE Workshop, ICCAD, ISSS, and the Design Automation Conference, as well as books [24].

VI. CONCLUSIONS

As microprocessors have grown exponentially in power, the design techniques required to build embedded computing systems have fundamentally changed. Embedded system courses need to emphasize fundamental concepts and methodologies in order to prepare students to build the billion-transistor systems on silicon of the future. Paradoxically, an emphasis on principles mixed with practice should make embedded computing easier to teach, since it reduces the dependency of lectures and labs on specific hardware platforms and reduces the need for lengthy and costly hardware debugging. As embedded computing systems become larger, designers are forced to clarify architectures and design techniques, yielding a set of principles that can be used by educators to teach the design of these systems.

VII. FOR MORE INFORMATION

For an online discussion of this special issue, please visit the discussion website at http://ieee.research.umich.edu.

ACKNOWLEDGMENT

The first author would like to thank the Princeton students who lived through his attempts to refine his notions on embedded systems education. The authors would like to thank J. Staunstrup of DTU for his support of the three-week course, as well as all the students for their enthusiasm and patience.

REFERENCES

[1] T. Schultz, *C and the 8051: Hardware, Modular Programming, and Multitasking*, 2nd ed. Upper Saddle River, NJ: Prentice-Hall, 1998, vol. 1.
[2] C. Mead and L. Conway, *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley, 1980.
[3] P. Thoma and M. Fuchs, "Automotive electronics—A challenge for systems engineering," in *Proc. DATE '99*, 1999.
[4] J. B. Peatman, *Microcomputer-Based Design*. New York: McGraw-Hill, 1977.
[5] C. Kaner, J. Falk, and H. Quoc Nguyen, *Testing Computer Software*, 2nd ed. Boston, MA: Int. Thomson Comput. Press, 1993.
[6] A. H. Mebane IV, J. R. Schmedake, I.-S. Chen, and A. P. Kadonaga, "Electronic and firmware design of the HP DesignJet drafting plotter," *Hewlett-Packard J.*, pp. 16–23, Dec. 1992.
[7] *The National Technology Roadmap for Semiconductors*, Semiconductor Industry Association, 1997.
[8] G. Myers, *The Art of Software Tests*. New York: Wiley, 1979.
[9] B. Beizer, *Software Testing Techniques*, 2nd ed. New York: Van Nostrand, 1990.
[10] N. G. Leveson, "Software safety: why, what, and how," *Comput. Surveys*, vol. 18, no. 2, pp. 125–163, June 1986.
[11] N. G. Leveson and C. S. Turner, "An investigation of the Therac-25 accidents," *IEEE Comput.*, pp. 18–41, July 1993.
[12] G. Booch, *Object-Oriented Design*. Redwood City, CA: Benjamin/Cummings, 1991.
[13] A. M. Davis, *Software Requirements: Analysis and Specification*. Englewood Cliffs, NJ: Prentice-Hall, 1990.
[14] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Programming*, vol. 8, pp. 231–274, 1987.
[15] D. J. Hatley and I. A. Pirbhai, *Strategies for Real-Time System Specification*. New York: Dorset House, 1988.
[16] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese, "Requirements specification for process-control systems," *IEEE Trans. Software Eng.*, vol. 20, pp. 684–707, Sept. 1994.
[17] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1991.
[18] B. Selic, G. Gullekson, and P. T. Ward, *Real-Time Object-Oriented Modeling*. New York: Wiley, 1994.
[19] C. E. McDowell, B. R. Montague, M. R. Allen, E. A. Baldwin, and M. E. Montoreano, "Javacam: Trimming Java down to size," *IEEE Internet Comput.*, pp. 53–59, May/June 1998.
[20] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 2nd ed. San Francisco, CA: Morgan Freeman, 1997.
[21] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hardreal-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.
[22] W. Wolf, A. Wolfe, S. Chinatti, R. Koshy, G. Slater, and S. Sun, "Lessons from the design of a PC-based private branch exchange," *Design Automation for Embedded Systems*, vol. 1, no. 4, pp. 297–314, 1996.
[23] G. De Micheli and R. Gupta, "Hardware-software co-design," *Proc. IEEE*, vol. 85, pp. 349–365, Mar. 1997.
[24] J. Staunstrup and W. Wolf, Eds., *Hardware/Software Co-Design: Principles and Practice*. Boston, MA: Kluwer, 1997.
[25] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley, 1999.

Wayne Wolf (Fellow, IEEE) received the B.S., M.S., and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1980, 1981, and 1984, respectively.

He is a Professor of Electrical Engineering at Princeton University. Before joining Princeton, he was with AT&T Bell Laboratories, Murray Hill, NJ. His research interests include hardware/software codesign and embedded computing, VLSI CAD, and multimedia computing systems.

Dr. Wolf has been elected to Phi Beta Kappa and Tau Beta Pi. He is a member of the ACM and SPIE.

Jan Madsen received the M.S. degree in electrical engineering and the Ph.D. degree in computer science from the Technical University of Denmark, Lyngby, in 1986 and 1992, respectively.

He is currently an Associated Professor at the Department of Information Technology, Technical University of Denmark. His research interests include high-level synthesis, hardware/software codesign, and system specification, modeling, and synthesis for embedded computer systems. He has published more than 40 papers.

Dr. Madsen has served on several conference program committees, including ISSS, Codes/CASHE, FTRTFT, and Euromicro. He is Program Chair for CODES 2000.