A Senior-Level Course in Hardware–Software Codesign

Patrick Schaumont, *Senior Member, IEEE*

***Abstract*—Modern electronic system design makes extensive use of programmable architectures, and requires designers to consider hardware and software jointly in their design. A senior-level course named Hardware/Software Codesign provides a practical introduction to these complex system design issues. The challenge is to bring a subject, which is traditionally covered as a graduate-level course, to senior undergraduate students without overly narrowing down the scope, and without turning the course into an ad-hoc design project. The course combines an incremental, structured overview of hardware/software codesign with practical assignments that emphasize key concepts. This paper reviews the motivations for this course, the curriculum, the lab materials and tools used, and the results of the first offering of the course in fall 2006.**

***Index Terms*—Computer architecture, education, hardware design languages, logic design, modeling, simulation software.**

I. Introduction

In fall 2006, a senior-level (fourth-year) course named Hardware/Software Codesign was organized at Virginia Polytechnic Institute and State University, Blacksburg. The course is targeted at undergraduates and explores the possibilities of hardware/software codesign as a standard design technique.

Academics and industry have praised the virtues of hardware/software codesign for many years [1], [2]. However, neither has shown the willingness truly to change their ways. Academic curricula tend to show a strong bias to either software design or hardware design. Industry employs the graduates in separated hardware and software teams as well. This approach is not viable in the long term.

Modern applications, especially those in the embedded area, make extensive use of programmable architectures such as shown in Table I. The reasons for this are well known. Being programmable, these components offer shorter application development times and lower design costs, and they enable a smooth tradeoff between application–flexibility and performance. But the use of specialized programmable architectures also poses new design challenges. First, design engineers have to learn new programming models and tools. In addition, a solid understanding of the underlying target architecture is essential in order to write efficient programs. Table I demonstrates that for design with contemporary components, the design of hardware and software can no longer be kept separate.

TABLE I
Architecture Design Space of Contemporary Programmable Chips

Device	Programming
MPU (Microprocessor)	HLL
Multi-core (Multiple MPU in one package)	ad-hoc
FPGA Soft-core (MPU inside of FPGA)	HLL
DSP (Digital Signal Processor)	HLL, Assembly
ASIP (App Specific Instruction Set Processor)	HLL + HDL
FPGA (Field Programmable Gate Array)	HDL

HLL = High Level Programming Language (e.g. C programming)
HDL = Hardware Description Language

The objective of the senior-level course in hardware/software codesign is to help students think of Table I as a design space rather than as a collection of point solutions. Hardware/software codesign is well suited for this because it naturally deals with specialized programmable architectures. For example, hardware/software codesign recognizes that an application will be implemented as the sum of a fixed part in hardware with a flexible part in software. Hardware/software codesign is also concerned with the design of efficient interfaces between hardware and software, which is critical to obtain efficient programs. Thus, rather than studying each of the individual components in Table I, students will learn the fundamentals by studying the principles of hardware/software codesign.

The paper uses the following outline. Section II reviews important concepts in the field of hardware/software codesign, and clarifies the obstacles in developing a course on this topic at undergraduate level. From these observations, three principles are derived for a senior-level course on hardware/software codesign. Section III introduces the course topics and emphasizes the important theoretical concepts and course structures. The section also reviews the supporting course materials. Section IV reviews related work and Section V analyzes the results of this course in terms of measurable learning objectives (MLOs). Section VI concludes the paper.

II. Mapping Hardware/Software Codesign Into an Undergraduate Course

Hardware/software codesign is a design technique that jointly addresses the creation of hardware and software in a design flow. The objective of codesign is to optimize performance and resource-usage of a design. *Hardware* and *software* do not have strict definitions. As illustrated in Table I, hardware includes a broad range of programmable components while software covers the programs written for those components. Proposed almost two decades ago, and continuously researched since then, hardware/software codesign has evolved into a

Manuscript received June 28, 2007; revised September 10, 2007. First published March 14, 2008; last published August 6, 2008 (projected).This work was supported in part by NSF CCR 0644070.

The author is with the Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061 USA (e-mail: schaum@vt.edu).

Digital Object Identifier 10.1109/TE.2007.910434

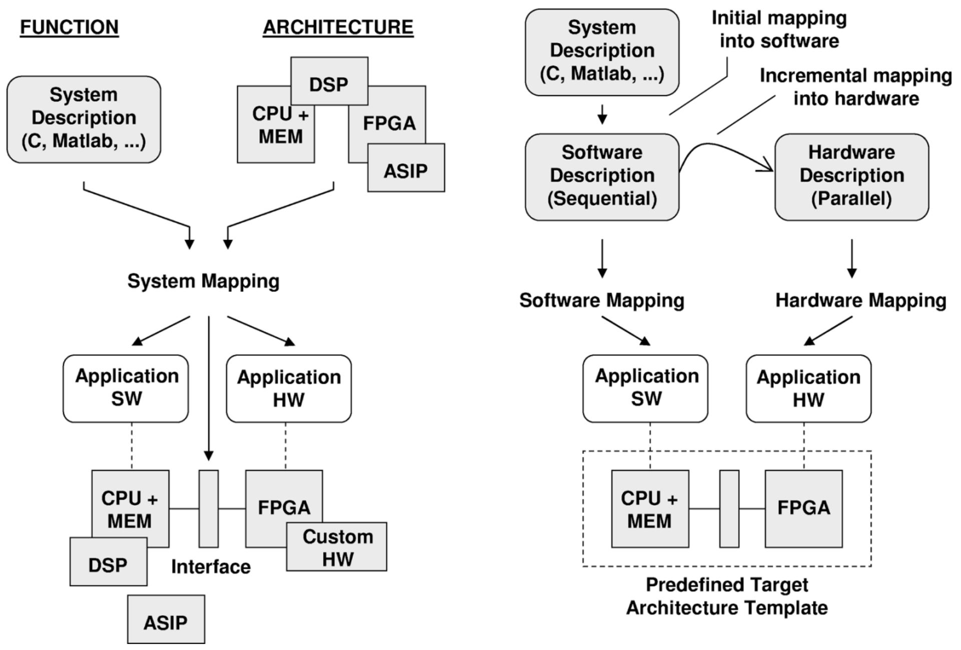
FUNCTION
ARCHITECTURE
System Description (C, Matlab, ...)
DSP
CPU + MEM
FPGA
ASIP
System Mapping
Application SW
Application HW
CPU + MEM
FPGA
Custom HW
DSP
Interface
ASIP
System Description (C, Matlab, ...)
Initial mapping into software
Incremental mapping into hardware
Software Description (Sequential)
Hardware Description (Parallel)
Software Mapping
Hardware Mapping
Application SW
Application HW
CPU + MEM
FPGA
Predefined Target Architecture Template

Fig. 1. (a) Function–architecture codesign. (b) Incremental hardware–software codesign.

well-established field. This section presents a short comparison between two mainstream approaches in codesign, highlighting the challenges faced in undergraduate course development. This analysis leads to the three principles used for development of this course.

Fig. 1 shows two approaches to hardware/software codesign. Fig. 1(a), *function/architecture codesign*, is the most general formulation. Also known as platform-based design [3], this technique starts from a system description of the application and a set of programmable components (called a platform). During system mapping, the application is mapped onto one or more programmable components, such that the resulting system architecture, the component interfaces, and the component programs are obtained. Platform-based design is a broad design technique and covers a wide range of architectures and applications. However, system-mapping is a designer-intensive and complex task, and there are currently no tools available that target design refinement for an arbitrary selection of programmable components.

Incremental hardware/software codesign, shown in Fig. 1(b), assumes that the target architecture template is fixed – for example a microprocessor with a field-programmable gate array (FPGA) attached to the processor bus. The system is mapped initially onto a single programmable component, and next redistributed incrementally over multiple components in order to obtain better overall system performance. Coprocessors and application-specific instruction-set processors are often designed in this manner [4]. The incremental nature of this process makes it easier to manage than function/architecture codesign. In addition, tools can automate large parts of the design process [5]. However, because of the predefined architecture template, the traversed design space is smaller.

The objective of the senior-level course in hardware/software codesign is to teach the fundamentals that govern the design flows in Fig. 1. In order to ensure that this material is manageable for senior-level students, the course adopted three design principles, which are discussed next.

A. Principle 1: Teach Incremental Concepts

In contrast to professional engineers in industry, students have limited upfront experience. This relative inexperience makes the flow of Fig. 1(b) easier to handle than Fig. 1(a). Indeed, with incremental hardware/software codesign, students can start from a concrete, working implementation, and observe the effect of small, incremental modifications. In contrast, function/architecture codesign requires more insight and experience. For example, the technique provides no suggestion on how to partition the system description initially over programmable components.

B. Principle 2: Simplify the Design Semantics

General-purpose codesign models for hardware and software span a broad range of abstraction levels. “Software” includes everything from a small dedicated assembly program up to a multithreaded application on top of an embedded operating system. “Hardware” includes everything from an instruction-set model of a microprocessor up to a low-level event-driven model of an asynchronous interface. Obviously this implies a wide range in design semantics. Since the focus in hardware/software codesign is on the interfaces between hardware and software, the course restricts hardware models and software models to have the following properties.

- Software is captured by single-thread sequential C programs, optionally extended with inline assembly code.
- Hardware is captured by cycle-based register-transfer level models. Note that this restricts hardware targets to synchronous components.

A rigorous application of this distinction then leads to two dual design views, illustrated in Table II. As a result, hardware and software can be used to address each others’ weaknesses.

TABLE II
DUALISM BETWEEN HARDWARE AND SOFTWARE

Hardware	Software
Parallel Execution	Sequential Execution
Design on a Fixed Time Budget	Design on a Variable Time Budget
→ Time constraints are easy	→ Time constraints are hard
Design using Variable Resources	Design using Fixed Resources
→ Area constraints are hard	→ Area constraints are easy
Programs yield models	Programs yield implementations
Flexibility is hard	Flexibility is easy
Data-intensive processing	Control-intensive processing

What is difficult with a hardware design approach is often easy with a software design approach and vice-versa.

Recent trends in design have suggested unifying hardware and software in a single language (such as C or SystemC). This unification makes sense from an industrial perspective. For educational purposes, the unification instead blurs the dualism between hardware design and software design. In addition, Edwards points out that moving hardware and software into a single syntax (in C) does not unify the semantics of hardware and software as well [6]. Therefore, the course makes explicit distinction between hardware models and software models. In fact, part of the lectures are spent discussing techniques to convert software models into hardware models and vice-versa.

C. Principle 3: Simplify the Tools

The tools required to simulate hardware/software models, and to compile or to synthesize them into an implementation, can require a steep learning curve. In addition, the current market of tools for hardware/software codesign, and more recently for electronic system level (ESL) design [7], is very dynamic. Other than standard modeling languages [like C, very-high-speed integrated circuits hardware description language (VHDL)], no generally adopted industry-standard design tools exist.

Therefore, the course uses an open-source cosimulation environment, called GEZEL [9], [10]. GEZEL offers cycle-based hardware modeling [based on finite state machine with datapath (FSMD) semantics] in combination with software modeling using instruction-set simulation. GEZEL is currently being used in educational projects at Virginia Tech, Katholieke Universiteit Leuven, Belgium, and the Denmark Technical Institute, Copenhagen, Denmark. The environment provides direct support for the simplified design semantics mentioned earlier. The strength of GEZEL is its simplicity; in fact only a single lecture is spent on the use of the tool and the creation of GEZEL models. At the same time, GEZEL can be easily extended with new cosimulation interfaces.

The next section explains the actual course content, which was devised following the principles outlined in this section.

III. COURSE DESIGN

Since hardware/software codesign is a relatively new technique, there is no generally accepted list of topics to be covered. This section explains the reasoning behind the course content, and elaborates upon several core topics. Table III lists the topics covered in the course. There are four major parts: 1) basic concepts; 2) fundamentals of custom architecture design; 3) methods to map applications into architectures; and 4) recent developments in codesign.

TABLE III
COURSE TOPICS

1. Basic Concepts	• Concurrency versus Parallelism
	• Control Flow versus Data Flow
	• Function versus Architecture
2. Custom architectures	• Finite-state-machine with datapath
	• Micro-programmed Architectures
	• Coprocessors: internals and interfaces
3. Design Methods	• Digital Signal Processing/ processors
	• Memory organization
4. Recent Developments	• Codesign with FPGAs
	• ASIPs
	• Synthesis of C into custom hardware

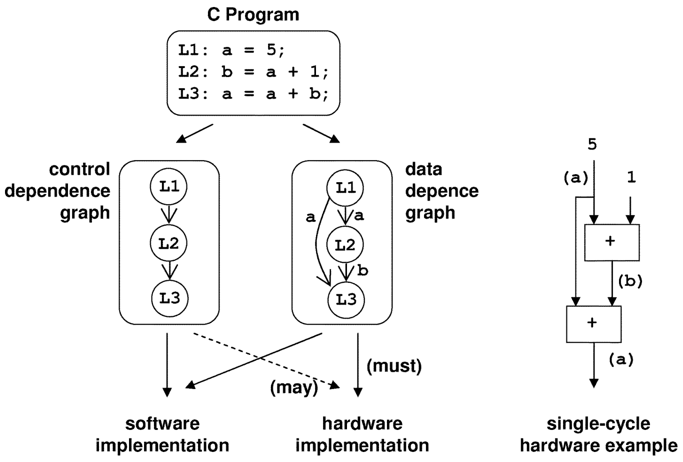

Fig. 2. Control-dependence graphs and data-dependence graphs help in analyzing software–hardware correspondence.

A fundamental idea in an incremental codesign flow is that hardware and software are two alternative implementation styles: any given system behavior that is captured as a software program can also be expressed as a hardware architecture. This observation, made by Madsen in [8], is the key to unifying hardware and software design topics in a single course. Even though the observation itself seems trivial, its implementation can be tricky due to the dualism between hardware and software. For example, given a function in C, it is not directly obvious how that function can be expressed as a cycle-based hardware model. Therefore, the initial discussions in the course aim to convince the students of the equivalence of hardware and software, and to make them comfortable moving from one world to the other. The following subsections will present three of these discussions, including data dependence and control dependence, concurrency and parallelism, and the design of custom architectures.

A. The Importance of Data Dependence

The first concept is the distinction between control dependence and data dependence. As illustrated in Fig. 2, any C program can be deconstructed into a control dependence graph and a data dependence graph. A control dependence graph represents each C statement as a node and each control transition as an edge. A data dependence graph maps each C statement as a node and each production/consumption of a variable as an edge. This analysis can be quickly understood by students and applied to various C programs, including programs with simple control structures. The importance of being able to *analyze* the

TABLE IV
CUSTOM ARCHITECTURE DESIGN SEQUENCE

Level	Hardware	Software	Limitations
Custom Datapath	Single-cycle design	none	No decision making
FSMD	Multi-cycle design	none	No complex decision making
Micro-engine	Multi-cycle design	Microprogram	Hard to pipeline and to program
RISC	Instruction-set architecture	C program	No customization using codesign
SoC	ISA with custom hardware	C program	NA

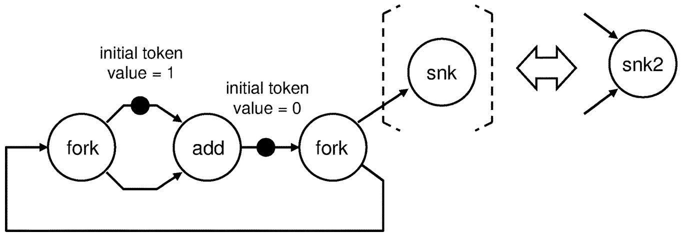

Fig. 3. Example assignment on concurrency: Students have to transform the SDF graph so that it concurrently generates two elements from the Fibonacci series rather than a single element.

data and control dependencies in a C program can hardly be overstated. Data dependencies are a genuine property of a specification, and will appear in any implementation of that specification. Control dependencies, on the other hand, are artificial. They are induced by the underlying sequential machine model from C. Consequently, an easy way of analyzing a C program with a parallel hardware implementation in mind is to focus on the data dependencies and to "ignore" the control dependencies. Fig. 2 illustrates the obvious correspondence between the data dependence graph of the example C program and a single-cycle hardware implementation of that program. In the course, students learn how to translate scalar C programs (including control flow statements) to hardware structures.

B. The Importance of Concurrency

The second important concept is the distinction between parallelism and concurrency. Both of these express the concept of simultaneous operation, but with an important difference. Parallelism relates to simultaneous activities of distinct components (such as the parallelism between two microprocessors). Concurrency, on the other hand, relates to design specifications, and expresses activities that potentially can execute simultaneously. Concurrent specifications are generally accepted to be better suited for system design than sequential specifications. Concurrency is important for the codesign course, because the combination of hardware and software leads to a concurrent system model. However, the course discusses concurrency separately from designing hardware and software models. Concurrency is studied by means of the synchronous dataflow (SDF) model of computation [11].

An example of an SDF graph, which is given as an assignment to the students, is shown in Fig. 3. The SDF model is a classic concurrent system model that maps functionality as a number of concurrent actors, which scan their inputs for the availability of tokens (input values). If a valid input is available, the actor will fire and produce a new token at its output. The fork actor duplicates a single input token on both outputs, while the add actor merges two tokens into a single one holding the sum of the input tokens. The communication channels in Fig. 3 are infinite queues that do not loose data. The model also contains two initial tokens, which hold the value of the first and second Fibonacci numbers. As tokens circulate around, the model generates the Fibonacci number series (0, 1, 1, 2, 3, 5, ...) into the single-input sink (snk) actor.

The objective of the assignment in Fig. 3 is to transform the graph so that it can generate two subsequent Fibonacci numbers for each iteration of initial tokens around the dataflow model. To solve this, students must replace the snk actor with a dual-input sink actor (snk2) and then unfold the graph. The simplicity and elegance of the SDF model allows students to focus on the concept of concurrency.

C. The Design Space of Custom Architectures

After a discussion of control/data dependencies, and of concurrency, the course addresses a core topic: a systematic overview of custom architectures. The objective is to gradually build up a system-on-chip architecture by considering increasingly complex architectures. This discussion crosses the boundaries of hardware and software. Indeed, the typical "hardware" textbook ends with a showcase of datapaths and state-machine controllers, while the typical embedded-software textbook starts from an microprocessor-based architecture with peripherals. However, from the viewpoint of hardware/software codesign, the architecture space is a continuum.

Table IV lists the sequence of architectures discussed in the course, in each case pointing out the "hardware" part, the "software" part, and the limitations that make designers abandon that architecture for a more sophisticated one. The following enumeration illustrates the line of thought in this lecture sequence.

The starting point of the discussion is the single-cycle custom datapath, which is fixed and which does not support control (decision making) very well. FSMD models express control and data processing separately [12], and can resolve these control issues. However, the FSMD model is not very good at handling complex decision making. Two well-known problems in this area are state explosion and awkward exception handling.

The next step is to initiate a discussion on microprogramming, which provides a more systematic design for the next-state logic in a controller. The micro-engine is the first programmable architecture covered in the course. It is remarkable that a design technique, invented 50 years ago by Maurice Wilkes, still plays a role in the systematic development of hardware/software codesign. But the micro-engine is also not without design complexities: as students can experience during an assignment, introducing pipeline stages in either the datapath or the microprogrammed controller complicates the design of microprograms.

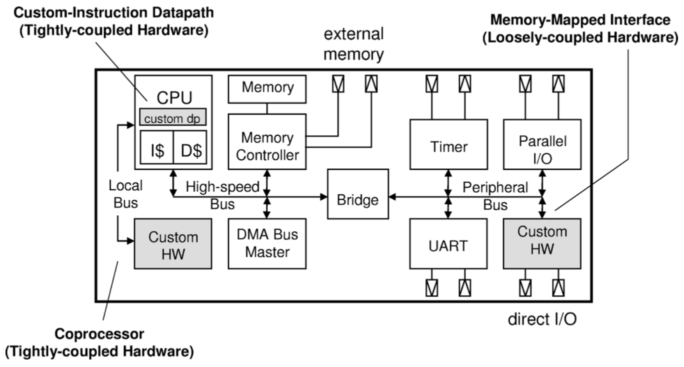

Fig. 4. System-on-chip architecture with interfaces for tightly-coupled and loosely-coupled hardware acceleration units.

This observation then leads to a machine that handles systematic pipelining very well: the reduced instruction set computer (RISC) processor. At this point in the course, the detailed behavior of hardware is abstracted out and replaced with the instruction-set architecture of the processor. The discussion on RISC resolves the pipelining issue, and also provides a starting point for hardware/software interfaces. These interfaces are discussed in context of the final custom architecture, the system-on-chip (SoC).

Different types of hardware/software interfaces can be introduced in the context of the SoC architecture. The interfaces can be classified based on the nature of the interaction between hardware and software (loosely-coupled versus tightly-coupled, memory-mapped versus port-mapped). Some advanced hardware–software interfacing mechanisms, such as direct-memory-access (DMA) and interrupts, are at present not included in the course material. The course discusses the interfaces illustrated in Fig. 4, which include a loosely-coupled memory-mapped interface typically used for peripherals, a local-bus interface for coprocessors, and a custom–instruction interface for tightly-coupled custom datapaths.

During the systematic development of hardware–software codesign, students use the GEZEL environment to simulate all these different forms of custom hardware and software.

D. Course Integration and Prerequisites

The course is part of the undergraduate computer-engineering curriculum at Virginia Tech. By their senior year, computer engineering students have taken a course on introductory digital design (covering combinational and sequential logic) and on microprocessor interfacing. The hardware–software codesign course for seniors uses these basic courses as prerequisites. Thus, students do not need to be familiar with hardware-description languages, or with advanced embedded software programming before entering the codesign course. The codesign course focuses on the transition area between hardware and software, and avoids overlap with existing courses in advanced computer engineering.

IV. Related Work

As pointed out above, hardware/software codesign is a novel topic at the undergraduate level. An insightful observation is made by Grimheden and Torngren on the didactics of embedded systems education [13], and their conclusions are valid for hardware/software codesign courses as well. They observe that present embedded systems education is done on a thematic, application-driven basis, often motivated by practical needs of industry. They also observe that embedded systems courses tend to be project driven and are taught in an interactive manner. This approach is in contrast to classic academic subjects such as calculus, for which there exists a broad consensus on the structure of the field, and which can use a disciplinary (rather than a thematic) approach.

An example of a graduate-level course that builds along the "platform-based" codesign model [see Fig. 1(a)] is Sangiovanni–Vincentelli's at the University of California, Berkeley [14]. However, teaching undergraduate-level codesign with complete architectural freedom remains difficult. Therefore, undergraduate courses in codesign tend to fix the platform upfront, and work along an incremental codesign model [see Fig. 1(b)]. An example of an early course that relied on the combination of a microprocessor and an FPGA is by Pottinger [15]. Contemporary FPGA technology enables users to combine a microprocessor and custom hardware. Therefore, more recent courses can migrate completely into FPGA, as described by Chamberlain [16] and by Bieser [17]. A hardware/software codesign course can also be formulated as a capstone project, such as for example that proposed by Klenke [18].

At present, education in hardware/software codesign faces two problems: the lack of an adequate textbook, and the lack of easy-to-use tools. The lack of textbooks may be caused by the absence of a generally agreed-upon structure for a codesign curriculum. Vahid and Givargis wrote one of the few recent undergraduate-level codesign books [19]. An older example is the book by Gajski [20]. Because of the lack of textbooks, students in the senior-level codesign course described here make use of instructor-provided course notes. A second problem is the lack of easy-to-use tools. Most of the abovementioned courses rely on standard hardware description languages (VHDL, Verilog) in combination with C programming. Since HDLs are very different from C, this complicates the learning process [21]. By using cycle-based modeling using FSMD semantics, GEZEL models avoid many of these problems.

V. Results

At the end of the course, students were asked to comment on the MLOs which are defined for this course. The five MLOs are listed in Table V. For each MLO, the 14 students that returned their feedback form (of the 17 students in the class) could rank the MLO from "Strongly Disagree" to "Strongly Agree."

Students were most positive about the first two MLOs, which reflect the understanding that hardware and software are equivalent forms (as discussed in Section III). They were less positive about the last two MLOs, which relate to the practical design aspect of hardware/software codesign. In hindsight, this impression seems to be caused by covering too much material, a comment some students made offline. In terms of the outline shown in Table III, this implies that parts 3 and 4 of the course should concentrate on the use of hardware–software interfaces rather than on system-level aspects.

TABLE V
MEASURABLE LEARNING OBJECTIVES

	Number of students that			
Measurable Learning Objective	Strongly Disagree	Disagree	Agree	Strongly Agree
Student could analyze and explain data- and control flow of a C program	0	1	5	8
Student could transform simple C programs into cycle-based hardware descriptions and vice versa	0	1	3	10
Student used simulation software to co-simulate C programs with cycle-based hardware descriptions	0	2	5	7
Student designed memory-mapped/ interrupt-driven interfaces to implement hw/sw communication	0	0	10	4
Student identified performance bottlenecks across hw/sw and optimize them	0	2	7	5

VI. Conclusion

The development of this undergraduate level course in codesign was motivated by the present evolution in the architecture landscape. The capabilities of modern architectures, such as million-gate FPGAs and multicore SoCs, are rapidly outgrowing the content of present software-oriented and hardware-oriented courses. In addition, each new architecture can no longer be taught separately because there are too many and too great a variety of them. Therefore, this course attempts to approach hardware/software codesign as a structured field, rather than an ad-hoc review of solutions.

Simplifying the design semantics and tools, and covering the subject in an incremental approach are both crucial to tackling the complexity of the subject and presenting it at undergraduate level. The idea that hardware and software can be treated as equivalent is the key to exploring various embodiments of hardware platforms, software applications, and frequently-used hardware/software interfaces.

Acknowledgment

The author would like to thank the support of W. Qin (SimIt-ARM's author) for tools during the course and the students who took the course during its first offering for their diligence and effort.

References

[1] G. de Micheli, R. Ernst, and W. Wolf, *Readings in Hardware/Software Codesign*. Norwell, MA: Morgan Kaufmann Silicon Series, Elsevier, 2001.

[2] W. Wolf, "A decade of hardware/software codesign," *Computer*, vol. 36, no. 4, pp. 38–43, Apr. 2003.

[3] A. Sangiovanni-Vincentelli and G. Martin, "Platform-based design and software design methodology for embedded systems," *IEEE Des. Test Comput.*, vol. 18, no. 6, pp. 23–33, Nov. 2002.

[4] C. Rowen*, Engineering the Complex Soc: Fast, Flexible Design with Configurable Processors*. Englewood Cliffs, NJ: Prentice-Hall, 2004.

[5] F. Sun, S. Ravi, A. Raghunathan, and N. Jha, "Custom-instruction synthesis for extensible-processor platforms," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 23, no. 2, pp. 216–228, Feb. 2004.

[6] S. Edwards, "The challenges of synthesizing hardware from C-like languages," *IEEE Des. Test Comput.*, vol. 23, no. 5, pp. 375–386, Sep./Oct. 2006.

[7] D. Densmore, A. Passerone, and A. Sangiovanni-Vincentelli, "A platform-based taxonomy for ESL design," *IEEE Des. Test Comput.*, vol. 23, no. 5, pp. 359–374, Sep./Oct. 2006.

[8] J. Madsen, J. Steensgaard-Madsen, and L. Christensen, "A sophomore course in codesign," *Computer*, vol. 35, no. 11, pp. 108–110, Nov. 2002.

[9] P. Schaumont and I. Verbauwhede, "A component-based design environment for electronic system-level design," *IEEE Des. Test Comput.*, vol. 23, no. 5, pp. 338–347, Sep./Oct. 2006.

[10] P. Schaumont, Hardware/Software Codesign Toolkit, 2007 [Online]. Available: http://rijndael.ece.vt.edu/codesign

[11] E. Lee and D. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, Jan. 1987.

[12] F. Vahid, "RTL design," in *Digital Design*. New York: Wiley, 2006.

[13] M. Grimheden and M. Torngren, "What is embedded systems and how should it be taught?," *ACM Trans. Comput. Syst.*, vol. 4, no. 3, pp. 633–651, Aug. 2005.

[14] A. L. Sangiovanni-Vincentelli and A. Pinto, "An overview of embedded system design education at Berkeley," *ACM Trans. Comput. Syst.*, vol. 4, no. 3, pp. 472–499, Aug. 2005.

[15] H. J. Pottinger, "Hardware-software co-verification in an undergraduate laboratory," in *Proc. IEEE Int. Conf. MicroElectronics System Education*, Arlington, VA, Jul. 1999, pp. 41–42.

[16] R. Chamberlain, J. Lockwood, S. Gayen, R. Hough, and P. Jones, "Use of a soft-core processor in a hardware/software codesign laboratory," in *Proc. IEEE Int. Conf. MicroElectronics System Education*, Anaheim, CA, Jun. 2005, pp. 97–98.

[17] C. Bieser, K. D. Muller-Glaser, and J. Becker, "Hardware/software co-training lab: From VHDL bit-level coding up to case-tool based system modeling," in *Proc. IEEE Int. Conf. MicroElectronics System Education*, Anaheim, CA, 2005, pp. 134–135.

[18] R. H. Klenke, J. Tucker, and J. Blevins, "A new hardware/software codesign environment and senior capstone design project for computer engineering," in *Proc. IEEE Int. Conf. MicroElectronics System Education*, Anaheim, CA, 2003, pp. 66–67.

[19] F. Vahid and T. Givargis*, Embedded System Design – A Unified Hardware/Software Introduction*. New York: Wiley, 2002.

[20] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1994.

[21] R. J. Duckworth, "Embedded system design with FPGAs using HDLs (Lessons learned and pitfalls to be Avoided)," in *Proc. IEEE Int. Conf. MicroElectronics System Education*, Anaheim, CA, 2005, pp. 35–36.

[22] P. Athanas and C. Patterson, "A holistic approach towards a unified CpE laboratory platform," in *Proc. IEEE Int. Conf. MicroElectronics System Education*, San Diego, CA, Jun. 2007, pp. 73–74.

Patrick Schaumont (M'97–SM'06) received the B.S. degree in electronic engineering from Hogeschool Gent, Gent, Belgium, the M.S. degree in computer science from Rijksuniversiteit Gent, Gent, Belgium, and the Ph.D. degree in electrical engineering from the University of California, Los Angeles.

He is an Assistant Professor of Computer Engineering at Virginia Polytechnic Institute and State University, Blacksburg. His research interests include design methods and architectures for embedded systems with an emphasis on demonstrating new methodologies in practical applications.