Computer-Aided Hardware-Software Codesign

Most digital systems consist of a hardware component and software programs that execute on the hardware platform. Obviously, a system can deliver higher performance when we tune the hardware to its software applications and vice versa. Today's novel architectures and the possible use of computer-aided design tools have created new opportunities to find solutions to codesign problems. This survey addresses this challenge, considers different architectures and their uses, and reports on the status of CAD codesign tools, with particular reference to simulation and synthesis.

Giovanni De Micheli

Stanford University

Though skillful digital designers have solved hardware-software codesign problems for many years, three recent developments have stimulated renewed interest in this topic.

First, today's computing systems deliver increasingly higher performance to end users. This may require architectural support for operating systems or particular hardware features to expedite application-specific software programs. At the same time, cost considerations have exploited the reprogrammability of the software component of digital systems to support product evolution. Second, new architectures based on programmable hardware circuits can accelerate the execution of specific computations or emulate new hardware designs.[1] Configuring and executing programs on these architectures exploit the synergy between hardware and software. Third, recent progress in synthesis[2] and simulation tools for hardware circuits has paved the way for integrating CAD environments for codesign of hardware-software systems.

Background

Since hardware-software codesign techniques can be applied in a vast number of areas, I've tried here to coarsely classify the specific problems and solutions in each sector. I consider the design of digital systems, or the digital component of electromechanical and/or mixed-signal systems, distinguishing general-purpose computing systems from dedicated computing and control systems. The former systems support generic software application programs and may range in size from palmtop computers to supercomputers. Thus, a wide disparity of operating system and programming languages may be available in this domain. The latter group of digital systems performs computations dedicated to some applications such as signal processing systems and embedded controllers. Again, the size of these systems may vary widely.

General-purpose and dedicated computing systems may consist of a variety of units, including standard instruction-set processors and/or coprocessors such as mathematical coprocessors for floating-point computation. Other coprocessors may be dedicated to subtasks, such as memory management or cache coherence handling in multiprocessor systems. Some dedicated computing and/or control systems may consist of one application-specific processing and control unit only.

Table 1 lists different approaches to the design of coprocessors and/or application-specific dig-

0740-7475/94/$04.00 © 1994 IEEE

Table 1. Trade-offs in different design approaches.

Trade-off	Standard coprocessor	Core coprocessor	ASIP	ASIC
Performance	Medium	Medium	High	Highest
Power	High	Medium	Medium-low	Lowest
Flexibility	Medium	High	High	Low
Design time	Low (software)	Medium (hardware, software)	Highest (hardware, software)	High (hardware)

ital units. At one end of the spectrum, application-specific integrated circuits (ASICs) provide a dedicated hardware solution, which usually can yield the best performance for the task at the expense of higher design time and cost. Dedicated application-specific instruction processors (ASIPs) contain an instruction set chosen to match the application, providing high performance as well as a desirable programmability feature. At the opposite end of the spectrum, a software-oriented solution consists of executing the application-specific digital function as a software program running on a dedicated processor.

The implementation of the units in a digital system depends on the chosen technology. The different units may consist of individual ICs on a board or on a multiple-chip module. Alternatively they can be integrated on a single substrate. Several processors available today as core macrocells can be combined with application-specific logic circuits in a single chip. Figure 1 gives an example.

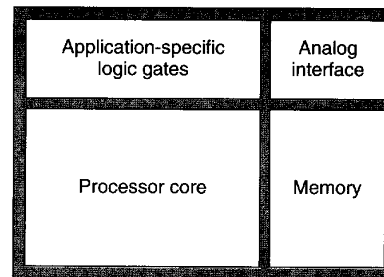

Figure 1. Example of an IC with a programmable core.

Codesign problems

Even though some CAD tools have been developed, most of today's hardware-software codesign problems still require hand-crafted solutions.

Instruction-set processors. These processors are the heart of information processing systems, and the frequent release of new microprocessors affects many strategic industrial decisions. Thus, the design of well-balanced, long-lasting processors is extremely important. Three issues in processor design require hardware and software considerations: instruction-set selection, cache design, and pipeline control.

Designers must determine the number and format of the instructions when selecting the appropriate instruction set. The compatibility of general-purpose instruction-set processors with other processors highly constrains this selection. On the other hand, this problem is less constrained and very relevant in determining the architecture of ASIPs. Instruction-set selection entails evaluating the benefits and costs of each instruction. Designers usually simulate a set of benchmark programs to determine the benefits, while the hardware required for instruction support determines the cost.[3]

The design and sizing of a memory cache require a match between circuit performance and the choice of the parameters and type of cache management algorithm (for example, update, invalidation). Traditionally, designers used simulation runs to determine cache sizes, but maintaining cache coherence is an important issue in multiprocessor design. For example, designers of the Flash multiprocessor at Stanford[4] chose an invalidation algorithm for cache management. A major concern in the design of Flash was avoiding the hardwiring of protocols into an ASIC chip. Therefore, the designers considered three programmable options for implementing the invalidation algorithm: a dedicated generic processor, programmable hardware, or an ASIP. They chose the last option because it allowed performance improvement in memory operations (as compared to a generic processor) and tailoring of the ASIP to speedily execute the specialized code while retaining some flexibility due to its programmability.

The design and control of a pipeline in a processor require that pipeline hazards be avoided, using either hardware or software techniques. A hardware mechanism would flush the pipelines, while a typical software solution would reorder instructions or insert no-operation instructions. The choice affects the overall processor performance. Furthermore, performance estimation is complex and requires appropriate models for both hardware and software.

Computing the most effective pipeline structure is thus a hardware-software codesign problem. The Piper synthesis program (developed at USC) is an example of a codesign CAD tool that addresses this issue.[5] It provides a means of

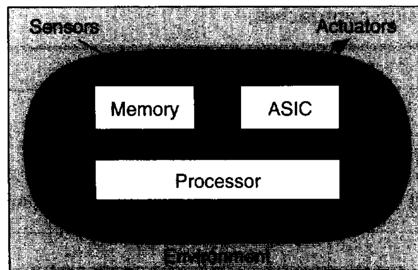

Figure 2. Scheme of the essential part of a mixed embedded system.

exploring the hardware-software trade-off and suggesting an effective implementation. In particular, Piper supports the automatic partitioning into pipe stages, pipeline scheduling, and determination of an appropriate instruction reordering that the corresponding back-end compiler should use to avoid hazards.

Signal processors. These processors play an important role in the telecommunication and consumer industries where high-performance, low-cost processors are indispensable. Several programmable DSPs can be produced in volume to keep the unit cost low. In addition, the use of signal processors in portable electronic devices requires low power consumption.

ASIPs present typical examples of codesign problems in this domain, particularly, instruction-set selection and code generation. The former problem is common to instruction-set processor design, but it is central to the design of ASIPs. Note that signal processors are often dedicated to one (or a few) function(s) and thus execute dedicated programs. Designers can choose the instruction set that supports fast execution of specific programs, while reducing power and area consumption. The optimization of performance relates to the shortening of the cycle time, latency, and maximization of the throughput.

The choice of instruction set affects the organization of the hardware. One avenue available to designers is using an integer linear programming formulation, as in the case of the Peas system (developed in Japan at the Toyohashi University of Technology).[6] Another approach, implemented in the Alchemy system[7] (codeveloped at Northwestern and Pennsylvania State universities), is based on an analysis of benchmark programs that are split into code segments, each of which is compiled optimally and weighted by its execution frequency. The final instruction set is a minimal cover of the instructions generated in compiling the segments.

The choice of an instruction set also affects the design of the corresponding compiler. Software compilers usually consist of three stages: a language-dependent front end, an intermediate code optimizer, and a code-generation back end. ASIP designers must then develop back ends that match the chosen instruction sets. Retargetable compilers address the problem of adapting their back ends to the desired instruction sets and/or specific hardware architectures.[8]

Research on retargetable code generation dates back to the development of the first compilers. Nevertheless, the design of ASIPs has renewed interest in this field. In particular, efficient code generation requires an accurate model of the ASIP data path. Novel code generators[9] use structural descriptions of the data path that can be generated by high-level hardware models. Thus, designers can model both hardware and software using high-level languages, and retargetable code generation can be seen as the counterpart of library binding (for hardware) in a codesign environment.

Embedded systems and controllers. These computing and control systems are dedicated to an application and may vary widely in size and scope. The most restrictive view of an embedded system is a microcontroller or a processor running a fixed program, such as a dishwasher controller. An example of a complex embedded system is an aircraft's guidance and control system. In general, embedded systems may have dedicated hardware as well as dedicated software running on one, or more, processors in addition to sensors and actuators to interact with the environment. (See Figure 2.)

Embedded systems often fall into the class of reactive systems. They are meant to react to the environment, by executing functions in response to specific input stimuli. In some cases, their functions must execute within predefined time windows. Hence they are called real-time systems. Examples of reactive real-time systems are pervasive in the automotive field (engine combustion control), in the manufacturing industry (robot controllers), and in the consumer and telecommunications industries (portable telephones).

A specific design problem for embedded systems is interfacing to the peripheral devices (sensors and actuators) according to the prescribed communication protocol. Usually a system designer will use standard parts for such devices, conforming to interfacing standards. Device drivers can be implemented in hardware (by ASICs) or by software routines. The choice depends on the number of available I/O ports of the processing unit and on the timing requirements for communication to peripheral devices. The Chinook design system supports the computer-aided design of embedded systems with peripherals. (See Chou, Walkup, and Borriello's article, this issue, p. 37.)

Embedded system designs are so numerous that the need for automating their design becomes very important. In addition, present embedded systems use simple processors (8-bit processors) because of the ease of manual design. Unfortunately, such a choice prevents leveraging the power of 32-bit (and 64-bit) processors now on the market, making

synthesis and validation of embedded systems a topic of recent high interest. The complexity of the problem stems from the interplay of the hardware and software components in embedded systems.[10]

Special architectures

Today's gap between hardware and software is smaller, because of the introduction of programmable hardware circuits, such as field-programmable gate arrays. Certain FPGAs implement arbitrary combinational or sequential circuits and can be configured by loading a local memory that determines the transistor interconnection.[1]

Applications of programmable hardware circuits are numerous. Boards populated by FPGA chips implement software functions (to speed up their execution) or to emulate hardware circuits (for prototyping). In the first case, the selection of the software segments to be implemented as programmable hardware is a codesign task. In the second case, executing software programs on programmable hardware boards can validate and optimize prototypes of complex digital systems. Since the hardware platform is retargetable, designers can explore hardware-software codesign trade-offs.

Software-execution acceleration. Certain bottlenecks in the execution of software programs limit performance. Examples are inner loops where sequences of operations are iterated. Coprocessors can help speed up the software execution of these routines, especially when they can be implemented in hardware to exploit the local parallelism. Unfortunately, hardware coprocessors are usually dedicated to specific tasks.

Coprocessors based on programmable hardware can provide comparable speedup capabilities (in some application domains) while being applicable to arbitrary software programs. Consider the programmable active memory (PAM),[11] which consists of a board of FPGAs and local memory interfaced to a host computer, as shown in Figure 3. Before executing a software program, a compiler compiles the critical portion of the program into patterns that configure the programmable board. This task takes about 50 μs. Then, the program executes, emulating the critical portions in hardware.

Successful applications of PAMs have been reported in different domains: cryptography, data compression, string matching, solution of equations modeling physical systems, and so on. Experimental results show a speedup of one to two orders of magnitude, as compared to a host's execution time.[11]

The major hardware-software codesign problem with PAMs consists of identifying the critical segments of the software programs, and compiling them efficiently to run on the programmable hardware. The former task is not yet automated for PAMs and is a current subject of research. The latter is based on hardware synthesis algorithms, and it benefits from performance optimization techniques for hardware circuits.

A specific project in this domain is the acceleration of digital system simulation by means of a specific hardware platform consisting of a tightly coupled processor and FPGAs. In this case, the simulator performs a preprocessing step in which the circuit model (to be simulated) is automatically partitioned and scheduled for execution on the platform to minimize runtime. Details appear in this issue in the Olukotun et al. article, p. 48.

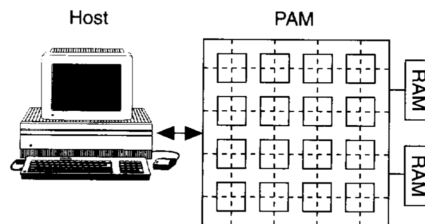

Figure 3. Programmable array memory.

Hardware emulation and prototyping. Computer-aided prototyping using programmable hardware boards is very useful when validating a hardware circuit before manufacture, and thus reducing the likelihood of an expensive redesign. Prototypes provide design engineers with more realistic data on correctness and performance than system-level simulations.[12]

Moreover, prototyping of complex digital systems including multiple hardware components and software programs appeals even more to designers. Prototyping allows tests of software program execution on hardware while retaining the capability to change the hardware (and software) implementation concurrently. Once finalized, the hardware configuration can be mapped onto a "hard" silicon implementation using synthesis systems[2] that accept as inputs hardware models compatible with those used by the emulation systems (for example, Verilog HDL models).

Validation and synthesis

CAD tools for mixed hardware-software systems, though limited, do exist for validation and synthesis needs.

Simulation of mixed systems. Circuit and/or system validation provides a reasonable certainty that a design is immune from errors and therefore ready for manufacturing. As circuits and systems become increasingly complex, validation becomes more important as well as more difficult. Designers use formal verification or simulation programs as well as prototyping techniques to validate designs.

Verification tools check the congruency of design representations or try to prove specific properties (for example, the existence of no-deadlock conditions). Today's designers apply verification in some restricted domains where regularity and abstraction can be used, but we are far from the stage where

we can fully verify complex hardware-software systems.

Simulation is a more traditional way of validating circuit correctness; we can examine a set of output responses to input stimuli. While simulation tools are widely available (even though only a few support mixed hardware-software representations), their use cannot ensure design correctness because only a limited number of input/output patterns can be analyzed. Nevertheless, simulation tools are still very useful for codesign. Hu et al. (this issue, p. 17) addresses the use of simulation in analyzing mixed systems in automotive electronic applications.

Designers can also use fast register-transfer-level simulators for hardware models in the Verilog HDL and VHDL languages. Similarly, register-transfer-level models for standard processors and cores have been developed in these languages. Thus, in principle it is possible to model a mixed system in a hardware description language for simulation purposes. In practice, register-transfer-level simulation is too slow to analyze systems executing software programs of reasonable complexity.

Coupling an instruction-set simulator for the processor (or core) to a register-transfer-level simulator for the application-specific hardware component lets designers perform cosimulation.[13] There are several possibilities for integrating simulators onto a simulation backplane. Specifically, when the processor under consideration is the same as the one used for simulation (for example, a Sparc core and a Sparc-based workstation), designers can coordinate the execution of the software code on the simulation host machine with the execution of the hardware simulator. In particular, using interprocess communication primitives helps.[14,15]

The difficulty in cosimulating hardware-software systems stems from their heterogeneous nature. The Ptolemy research design environment and simulator[16] for signal processing and communication-system design (developed at the University of California at Berkeley) particularly addresses the problem of heterogeneity in cospecification. The Ptolemy user can model a mixed system with different paradigms (dataflow and discrete-event models) including user-defined models. Ptolemy provides a simulation backplane in which the description in the different domains can be simulated and consistently interfaced, thus providing an efficient means for validating mixed systems.

Synthesis of mixed systems. Computer-aided synthesis of dedicated computing systems, or cosynthesis, will evolve naturally from existing hardware architectural synthesis methods.[2] A working hypothesis for cosynthesis is that the overall system can be modeled consistently and be partitioned, either manually or automatically, into hardware and software components. Application-specific circuits using existing hardware synthesis tools can implement the hardware component; the software component can be generated automatically to implement the function to which the processor is dedicated. Cosynthesis must also provide a means for interfacing and synchronizing the functions implemented in the hardware and software components.

System-level modeling. Modeling at this level plays a major role in supporting cosynthesis. Some proposed abstraction models can be grouped into major classes: 1) representation of system behavior by means of control/dataflow graphs (CDFGs)[2] and 2) cooperating finite-state machines (FSMs).

CDFGs abstract the behavior of a digital system as a set of tasks and a set of (dataflow or control flow) dependencies. Thus, this modeling style applies to both hardware and software. Indeed, CDFGs can be derived directly from hardware and software language models and have been used successfully to support hardware architectural synthesis and software compilation.

FSM models capture system states and transitions and apply to both hardware and software portions of a design. State charts[17] provide an example of a modeling style in terms of concurrent and hierarchical FSMs. SpecCharts[18] combine the FSM and programming formalisms. When systems are not modeled directly by means of FSMs, FSM models can be derived for either hardware or software languages.

Different execution models exist for cooperating FSMs. The usual notion of concurrent FSMs is based on the synchrony hypothesis (all FSMs have simultaneous transitions). Such a modeling style is more appropriate for hardware than for software, because of the unknown or uncontrollable delays associated with the execution of software fragments.

Due to the disparity of execution speeds between hardware and software implementations, Chou et al. (this issue, p. 37) proposes modeling hardware-software systems by FSMs that interact via events with possibly unbounded delays. This paradigm extends hardware synthesis and verification techniques to cope efficiently with the mixed systems.

A drawback of using cooperative FSM models for codesign is that the system partition into hardware and software is mandated (or constrained) by the initial system specification. Thus, research on hardware-software trade-offs using automatic partitioning techniques has been leveraging CDFG models.

System-level partitioning. Partitioning of the hardware and software components affects the overall system cost and performance. Hardware solutions may provide higher performance by supporting parallel execution of operations at the expense of requiring the fabrication of one (or more) ASICs. Software solutions may run on high-performing processors available at low cost due to high-volume production. Nevertheless, operation serialization and lack of specific support for some tasks may result in loss of performance. Thus, a system design for a given market may find its cost-effective implementation by splitting its functions between hardware and software.

Several researchers have investigated system-level parti-

tioning into hardware and software components.[15,19,20] Two approaches appear interesting: the synthesis of dedicated coprocessors for hardware speedup[20] and the migration of noncritical functions to software.[19] The two problems have complementary objectives: The former attempts to maximize performance, while the latter tries to minimize system cost, subject to performance constraints.

The Cosyma synthesis tool suite[20] (developed at the University of Braunschweig, Germany) partitions a system specification to speed up software execution by using a dedicated hardware coprocessor (to be synthesized). Cosyma describes the original system model as a software program in C*, which is an extension of the C programming language, to support performance constraints. A software implementation of the initial model is readily available (by compilation), but it may be delivering performance inferior to expectations. The C* compiler compiles the system model into a control/dataflow graph, and a partitioning algorithm identifies the computational bottlenecks and then migrates the corresponding functions to application-specific hardware. As an example of an application, designers expedited the chromakey algorithm for high-definition television (HDTV) on a Sparc processor by a factor of three. They identified a critical loop that takes 90 percent of the software runtime and fabricated an ASIC with 17,000 equivalent gates. Figure 4 shows details of Cosyma.

The Vulcan synthesis tool suite[19] uses a hardware model of the system and attempts to reduce the cost of its implementation by transferring noncritical operations to a standard coprocessor, such as an 8086 or R3000. In particular, Vulcan models systems in the HardwareC hardware description language, which has a C-like syntax but hardware semantics. The system model specifies performance requirements, in terms of latency and data rate constraints.

While a hardware implementation can be derived from the HardwareC model by using the Olympus synthesis tools[21] (developed at Stanford University), the cosynthesis approach is as follows. Vulcan compiles the system model into a control/dataflow graph and partitions it. This yields a set of software threads to be compiled and executed on the standard coprocessor and the specification of the remaining hardware circuits for synthesis as a netlist of logic gates. An Ethernet coprocessor satisfying the timing requirements of the CSMA/CD communications protocol has been partitioned by Vulcan into a mixed implementation using an R3000 processor. The implementation saved 20 percent more equivalent gates than in the case of a hardware implementation, allowing a smaller and cheaper ASIC gate array to be used. Figure 5 (next page) gives details of the Vulcan suite.

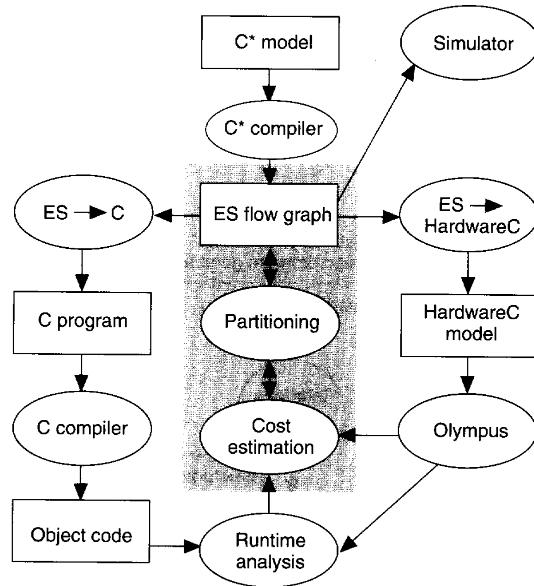

Figure 4. The Cosyma cosynthesis tool suite. (ESgraph is a form of CDFG.)

AT PRESENT, THE OVERALL CAD SUPPORT for hardware-software codesign is still primitive. Nevertheless, the potential payoffs make it an attractive area for further research and development. Several open problems still impede the rapid growth of the field. First and foremost is a need to define better abstract models for hardware-software systems and their environments as well as to develop consistent languages to express them. Possible solutions range from the extension of existing hardware and software languages to the use of new heterogeneous paradigms.

Cost and performance evaluation of mixed systems also plays an important role in driving partitioning and synthesis decisions. The problem is complicated by the remoteness of the abstract system models from the physical implementation.

In addition, the still-primitive cosynthesis and cosimulation methods leave plenty of room for improvement.

Last, but not least, methods for formal verification of properties of mixed systems would be extremely useful, because cosimulation may provide insufficient evidence of design correctness. Extending formal verification techniques to the hardware-software domain would thus be desirable.

Acknowledgments

NSF/ARPA, under grant MIP 9115432, and the Stanford Center for Integrated Systems supported this work.

Figure 5. The Vulcan cosynthesis tool suite.

References

1. S. Trimberger, "A Reprogrammable Gate Array and Applications," *IEEE Proc.*, IEEE, Piscataway, N.J., July 1993, pp. 1030-1041.
2. G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, N.Y., 1994.
3. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, Calif., 1990.
4. M. Horowitz and K. Keutzer, "Hardware-Software Codesign," *Proc. SASIMI (Synthesis and Simulation Meeting and International Interchange)*, Nara, Japan, 1993, pp. 5-14.
5. I. Huang and A. Despain, "High-Level Synthesis of Pipelined Instruction Set Processors and Back-End Compilers," *Proc. Design Automation Conf.*, Assoc. of Computing Machinery, New York, 1992, pp. 135-140.
6. A. Alomary et al., "An ASIP Instruction Set Optimization Algorithm with Functional Module Sharing Constraints," *Proc. Int'l Conf. Computer-Aided Design*, IEEE Computer Society Press, Los Alamitos, Calif., 1993, pp. 526-532.
7. B. Holmer and B. Pangrle, "Hardware/Software Codesign Using Automated Instruction Set Design and Processor Synthesis," *Proc. ACM Workshop on Hardware-Software Codesign*, ACM, Oct. 1992.
8. C. Liem, T. May, and P. Paulin, "Instruction-Set Matching and Selection for DSP and ASIP Code Generation," *Proc. European Design and Test Conf.*, CS Press, 1994, pp. 31-37.
9. P. Marwedel, "Tree-Based Mapping of Algorithms to Predefined Structures," *Proc. Int'l Conf. Computer Design*, CS Press, 1993, pp. 586-593.
10. W. Wolf and E. Frey, "Tutorial on Embedded System Design," *Proc. Int'l Conf. Computer Design*, CS Press, 1992, pp. 18-21.
11. P. Bertin, D. Roncin, and J. Vuillemin, "Introduction to Programmable Active Memories," in *Systolic Array Processors*, J. McCanny, J. McWhirter, and E. Schwartzlander, eds., Prentice Hall, Englewood Cliffs, N.J., 1989.
12. L. Maliniak, "Logic Emulator Meets the Demands of CPU Designers," *Electronic Design*, Apr. 1993.
13. R. Gupta, C. Coelho, and G. De Micheli, "Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components," *Proc. Design Automation Conf.*, ACM, 1992, pp. 225-230.
14. D. Becker, R. Singh, and S. Tell, "An Engineering Environment for Hardware-Software Cosimulation," *Proc. Design Automation Conf.*, ACM, 1992, pp. 129-134.
15. D. Thomas, J. Adams, and H. Schmit, "A Model and Methodology for Hardware-Software Codesign," *IEEE Design & Test of Computers*, Sept. 1993, pp. 6-15.
16. A. Kalavade and E. Lee, "A Hardware-Software Codesign Methodology for DSP Applications," *IEEE Design & Test of Computers*, Sept. 1993, pp. 16-28.
17. D. Harel et al., "On the Formal Semantics of State Charts," *Logic and Computer Science*, 1986.
18. S. Narayan, F. Vahid, and D. Gajski, "Translating System Specifications to VHDL," *Proc. European Design Automation Conf.*, CS Press, 1991, pp. 390-394.
19. R. Gupta and G. De Micheli, "Hardware-Software Cosynthesis for Digital Systems," *IEEE Design & Test of Computers*, Sept. 1993, pp. 29-41.
20. R. Ernst, J. Henkel, and T. Benner, "Hardware-Software Cosynthesis for Microcontrollers," *IEEE Design & Test of Computers*, Dec. 1993, pp. 64-75.
21. G. De Micheli et al., "The Olympus Synthesis System for Digital Design," *IEEE Design & Test of Computers*, Oct. 1990, pp. 37-53.

Giovanni De Micheli's biography, photograph, and address appear on p. 9 in this issue.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 153 Medium 154 High 155