The Softening of Hardware

Frank Vahid
University of California, Riverside

Programmable processors and IC fabrics are making once inflexible hardware "soft," blurring the line separating software and hardware. Such trends echo the early days of computing, when engineers viewed machines and their programs as single entities.

The first programmable computers were rather nasty beasts. These giant 1940s machines occupied entire rooms and consumed kilowatts of electricity. The computation problems engineers faced more often involved failed machine hardware components than the short and mostly unchanging programs engineers fed to those machines. Engineers viewed computers and their programs as unified entities.

Over the next decade, computers became more stable while programs grew more complex. Solving problems encountered during computing, or *debugging*, began to have less to do with removing heat-loving insects from the heated hardware components and more to do with finding and correcting logical flaws in the programs. Hence, the frequently changing programs, or *software*, became distinguished from the unchanging hardware on which they ran. A 1958 article may be the first to have used the term software.[1]

> Today the "software" comprising the carefully planned interpretive routines, compilers, and other aspects of automative programming are at least as important to the modern electronic calculator as its "hardware" of tubes, transistors, wires, tapes, and the like.

The software and hardware development fields evolved along separate paths through the end of the 20th century. We seem to have come full circle, however. The previously rigid hardware on which our programs run is softening in many ways. Embedded systems are largely responsible for this softening. These hidden computing systems drive the electronic products around us, including consumer products like digital cameras and personal digital assistants, office automation equipment like copy machines and printers, medical devices like heart monitors and ventilators, and automotive electronics like cruise controls and antilock brakes.

Embedded systems force designers to work under incredibly tight time-to-market, power consumption, size, performance, flexibility, and cost constraints. Many technologies introduced over the past two decades have sought to help satisfy these constraints. To understand these technologies, it is important to first distinguish the underlying embedded systems elements.

EMBEDDED SYSTEMS ELEMENTS

New technologies tend to earn names stressing their distinguishing traits, but in the rapidly changing computer world, such terms quickly become outdated. A personal computer that serves files to thousands of users isn't very personal, for example, and today's floppy disks are actually quite rigid.

This rapid outdating of terminology creates confusion. However, at this point in the evolution of embedded computing, we can at least distinguish among three elements that are often confused: processors, integrated circuit

0018-9162/03/$17.00 © 2003 IEEE

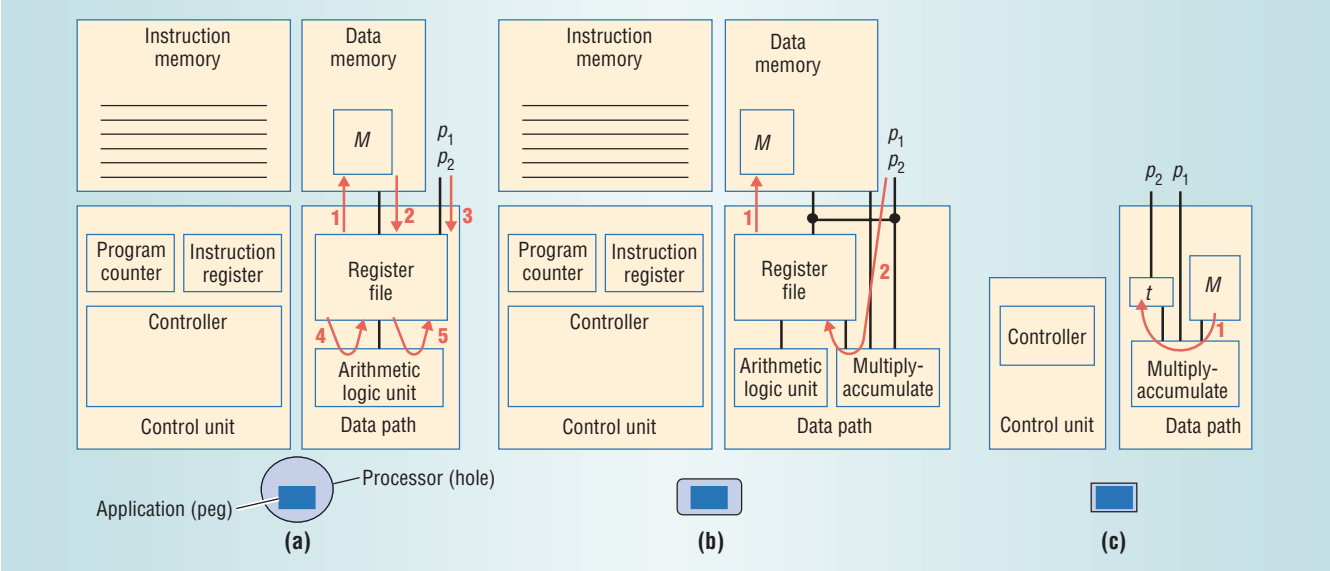

Figure 1. Processor types executing a simple application. (a) A general-purpose programmable processor requires five steps to implement a given statement; (b) a semicustom programmable processor, which is optimized for particular classes of programs, requires two steps; and (c) a custom processor requires only one step to implement the statement.

fabrics, and chips. We rely on these elements to convert our applications—the collections of algorithms that compose the computing behavior of our embedded systems—into real-world products.

Processors: Executing applications

A processor is a digital design capable of executing an algorithm and typically includes a data path and a controller. The data path includes basic digital components like registers that store data, functional units that transform data (for example, by adding or shifting data), and multiplexers and buses that move data. The controller includes registers and random logic gates that configure the data path for such stores, transforms, and moves—a large enough sequence of which we call an algorithm. Many processors are programmable, but many are not.

Square pegs and round holes: Programmable versus custom processors. "You can't fit a square peg into a round hole." Actually, given a large enough hole, you can. To bring their applications (the pegs) to life, embedded systems designers must map applications into processors (the holes). Processor types range from general-purpose to custom.

General-purpose processors are programmable, and are designed to execute nearly any application. Their data paths contain large register files and flexible arithmetic-logic functional units. Their controllers do not incorporate information about the application that the processor runs; instead, the controllers execute memory-stored instructions representing the application.

Mapping applications to general-purpose processors requires fast (measured in seconds) compilers and other mature tools. These processors are the holes big enough to fit any peg.

Designers can also use custom processors to execute applications. Custom processors can match the number, sizes, and interconnections of registers and functional units to the application to best meet performance and power constraints. They can encode the application into the controller, eliminating slow and power-hungry accesses to instruction memory.

Custom processors might include deep pipelines or numerous data-path functional units to achieve parallelism. The off-the-shelf custom processors that are available for many common application computations are referred to as coprocessors, accelerators, or peripherals. For other applications, designers must build their own custom processors. Custom processors are the holes cut to fit one peg shape only.

Figure 1 illustrates how three different processor types would implement a simple application requiring the processor to read an input stream from a port $p1$, perform a multiply-accumulate using an array of constants M, and output the final sum over port $p2$.

Figure 1a shows how a general-purpose processor would use the following sequence of steps stored in memory to implement the multiple-accumulate statement $t = t + M[i] * p1$, which would be enclosed in a loop that increments i.

1. Move i from the register file to the data memory address register.
2. Read $M[i]$ into the register file.
3. Store $p1$ in the register file.
4. Read $M[i]$ and $p1$ from the register file, multiply them, and store the result in the register file.
5. Read t and $M[i] * p1$ from the register file, add them, and store the result in t in the register file.

Figure 1b represents a semicustom programmable processor (described later), and Figure 1c represents a custom processor implementation. On the custom processor, the statement would execute in one step.

The advantages of general-purpose processors include their immediate availability, low cost (due to mass production), and simple design flow. Designers especially like general-purpose processors' flexibility—they can reprogram these processors in minutes and fix bugs during the last stages of product design. However, designers often need the advantages that custom-designed processors offer, including improved performance, lower power, and reduced cost (when manufactured in large quantities).

Such competing demands epitomize the embedded systems design problem. Broadly, designers must partition their applications to find a balance between general and custom solutions.

Two sizes don't fit all: Semicustom processors. Not long after designers began incorporating processors in embedded systems, processor makers introduced new classes of devices—such as digital signal processors (DSPs) and microcontrollers—that fell between the extremes of general-purpose and custom processors.

Some embedded computers digitize analog signals (such as audio and video signals), transform them, and send out new signals. Unlike the less dynamic files or user-input data that desktop computers process, analog signals tend to stream in and out rapidly. Programmable DSPs incorporate special instructions and supporting underlying hardware to efficiently handle signal streams and the common operations applied to them. For example, a programmable DSP might contain instructions for fast reads and writes of large arrays, single-cycle multiply-accumulate, or fast floating-point arithmetic.

Embedded computers that respond to events—a press of a button or a triggering of a sensor, for example—must sense that event and respond with a new event, such as turning on a light or opening a door. These computers have less need for streaming signal operations, requiring instead bit-level operations, efficient access to external wires, and low power consumption.

Programmable microcontroller architectures are tuned to such control behavior, often having narrow data paths (16, 8, or even 4 bits are common), simple functional units, registers directly connected to external pins, and extensive instructions for bit-level manipulation. They might also closely integrate timers, serial communication, analog-digital converters, and other common embedded control functions.

DSPs and microcontrollers represent early and widespread forms of the growing class of semicustom programmable processors, also known as application-specific instruction-set processors (ASIPs).[2-4] Semicustom processors are optimized for particular classes of programs, such as digital picture processing, network routing, or mobile communications. A picture-processing ASIP, for example, might include datapath components and special instructions to assist with picture compression.

Figure 1b represents a semicustom processor implementing the example application used in Figure 1a. This processor would execute the statement $t = t + M[i] * p1$ in just two steps:

1. Move i from the register file to the data memory address register.
2. Read t from the register file and $M[i]$ from data memory, use the multiply-accumulate (MAC) unit to compute $t = t + M[i] * p1$, and store the result in t in the register file.

No boundaries exist between processor types; the types simply represent general categories along a continuum from general to custom.

The embedded systems design problem requires partitioning applications to find a balance between general and custom solutions.

IC fabrics: Giving processors their style

An IC is an interconnection of transistors following one of several possible styles, or *fabrics*. Just as you can make shirts using different knits, you can build processors using different IC fabrics.

Programmable versus custom fabrics. Transistors are the fundamental digital entities that make up a processor's components—its registers, memories, functional units, and logic gates. How and when we compose these transistors depends on the IC fabric we use. Like processors, IC fabrics differ in terms of their customizability and generality.

A chip maker could custom compose transistors to implement the components of a processor exactly,[5] and then send the transistor design to a chip fabrication plant. The resulting chip would be a compact, fast, and perhaps even low-power implementation. If a processor requires AND and OR functions, for example, a designer could create a circuit with an AND and OR gate, as Figure 2 shows.

At the other extreme, a chip maker could build a set of interconnected modules and program each set to implement different components.[6,7] A module might be a small, 16-word memory. Storing the appropriate bits (that is, a configuration) in the memory—the traditional definition of *programming*—can implement a desired combinational function. The chip maker could develop an IC fab-

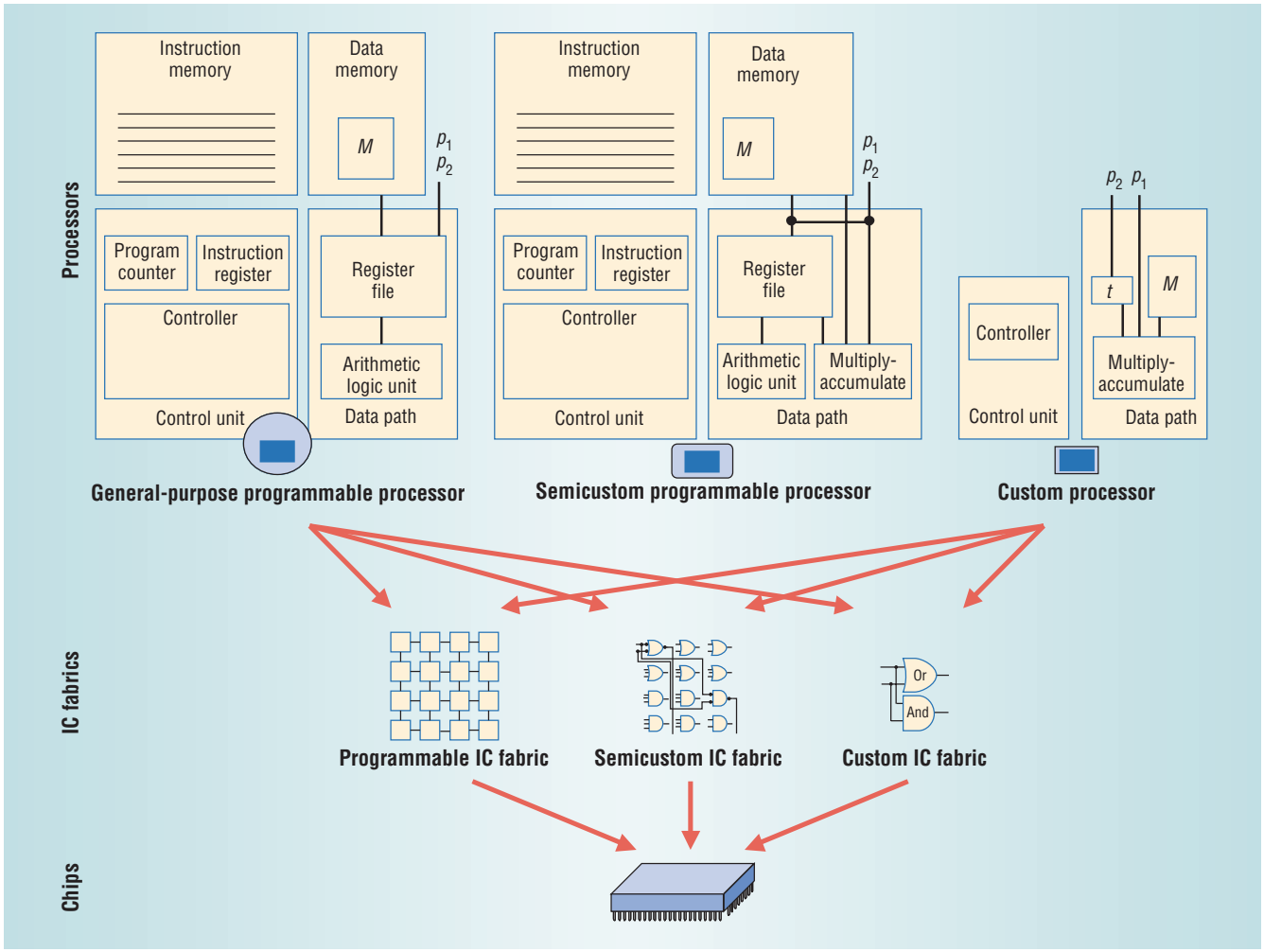

Figure 2. The relationships among processors, IC fabrics, and chips. Chips physically implement IC fabrics, which in turn implement processors. High-capacity chips can support multiple general-purpose, semicustom, and custom processors.

ric using many such combinational modules, plus register modules for storage, and use a programmable interconnect made up of multiplexers whose select line is controlled by a bit in another programmable memory to link the modules.

Because digital components consist of combinational logic and storage, we could map any set of such components onto this fabric, given adequate resources. Components can extend across modules, and connected components can exist on modules separated by large distances and connected through numerous programmable interconnects. The system will still work, although it will consume more power and perform less efficiently than a custom processor in which components are close together and connections are direct.

The most popular programmable IC fabric today is the field-programmable gate array. An FPGA is actually a programmable IC fabric consisting of programmable logic modules, storage, and interconnect, with little resemblance to an array of gates. The term is likely due to the preplaced transistor feature FPGAs share with gate array technology, which was popular when FPGAs first appeared.

The tradeoffs between custom and programmable IC fabrics are similar to those for processors. Designers like the immediate availability, simpler design flow, and late-change flexibility of programmable IC fabrics. However, they often need the performance, power, size, and cost advantages of custom fabrics.

Semicustom IC fabrics. Several semicustom IC fabrics have evolved to meet designers' needs for more options in between custom and programmable fabrics.[6]

Designing custom transistor circuits requires much effort to both avoid and fix mistakes: Place two transistors too close together or make one transistor too small, and the entire circuit might fail.

Using predesigned transistor circuits (libraries) of basic logic components, or *standard cells*, reduces this effort. A designer merely places these uniformly sized cells and routes wires to connect them before sending the design to a chip fabrication plant. The fabricated chip will likely have fewer errors than a custom IC fabric, thus reducing chip design and fabrication time from many months to perhaps just one or two.

A gate array IC fabric reduces chip design time further by providing transistors that have been prearranged into logic gate rows (arrays). A chip maker can thus premanufacture much of the chip, and

designers need only connect the gates, reducing chip design and fabrication time to perhaps just weeks.

Additional semicustom fabrics include *cell arrays*, which prearrange groups of cells in rows, so designers need only connect them. Another method of creating a semicustom fabric removes undesired connections rather than creating desired connections, perhaps with a laser. Such a fabric can be premanufactured, and the removal process may only take a week or even a day. Many additional semicustom fabrics exist and are evolving.

Chips: Anchoring IC fabrics to the physical world

A chip is the thumbnail-sized piece of silicon that physically implements IC fabrics, which in turn implement processors. The chip consists of many layers of elements forming transistors and wires.

Chip fabrication processes, also known as IC technologies, have steadily improved over the past several decades. The most prominent improvement in IC technology is the decreasing *feature size*—roughly defined as the size of the narrowest wire or transistor part. Feature sizes below 100 nanometers are becoming common.

For processors to work in the physical world, an IC fabric must appear on a physical chip. As Intel cofounder Gordon Moore predicted in the 1960s, chip transistor capacity has been doubling roughly every 18 months for several decades. Chip manufacturers can exploit the decreasing chip feature size predicted by Moore's law to pack in more transistors. Whereas early chips integrated tens or hundreds of transistors, current chips integrate tens or hundreds of millions of transistors.

A single high-capacity chip can support multiple general-purpose and custom processors, enabling today's high-functionality embedded systems. Several decades ago, a processor might have required numerous chips. The 1970s and 1980s saw the advent of microprocessors, general-purpose processors implemented either on one or just a few chips.

The term *application-specific integrated circuit* (ASIC) was introduced to refer to chips implementing custom processors, distinguishing them from microprocessor ICs. In the 1990s, designers introduced the term *microprocessor core*—a microprocessor appearing on a chip with other processors, rather than being on its own chip. *Core* has further evolved to refer to custom processors sharing a chip as well. Thus, a core is basically a processor—some are general-purpose, some semicustom, and some custom.

Although single chips have long integrated both custom and semicustom fabrics, high-capacity chips have recently begun to integrate programmable fabrics as well. Furthermore, designers can now use high-capacity chips to implement programmable fabrics on top of semicustom fabrics.

Figure 2 shows the relationships between processors, IC fabrics, and chips. Designers map processors to IC fabrics, and IC fabrics onto chips. Several processors often coexist on a single chip and the same chip can incorporate numerous IC fabrics.

A processor can utilize a single fabric or multiple fabrics, such as a custom fabric for the data path and a semicustom fabric for the controller. Countless possible mappings of processors to fabrics and fabrics to chips exist.

The soft bits we now download to a chip may represent much more than instructions.

SOFTER HARDWARE

The term "software" is usually used to represent the instructions to be executed on programmable hardware processors, or to contrast those instructions on a processor with the custom hardware processors, memories, and buses that complete a system. At the same time, software represents the soft bits—the zeros and ones—that configure the system to realize a specific application.

The uses of the term software are growing increasingly distinct, however, as soft bits often represent much more than instructions. In particular, the programmable IC fabrics of today's chips also have memories that must be configured, leading to the first cause of a blurring between software and hardware:

> Of the "soft" bits we download to a chip, some have the traditional role of representing instructions to be executed on a programmable processor, but others now represent processors themselves being mapped onto programmable IC fabrics.

The bits often represent custom processors, but they also can represent part or all of a semicustom processor or even a general-purpose processor being mapped to a programmable IC fabric.

Two additional trends are changing how we view software and hardware. *Reconfigurable computing* occurs during application execution, allowing an application to swap processors or parts of processors in and out of limited programmable IC fabric to perform the computations an application needs.

Systems with *tunable architectures* contain processors, memories, and buses with additional configurable features. A tunable architecture can reduce

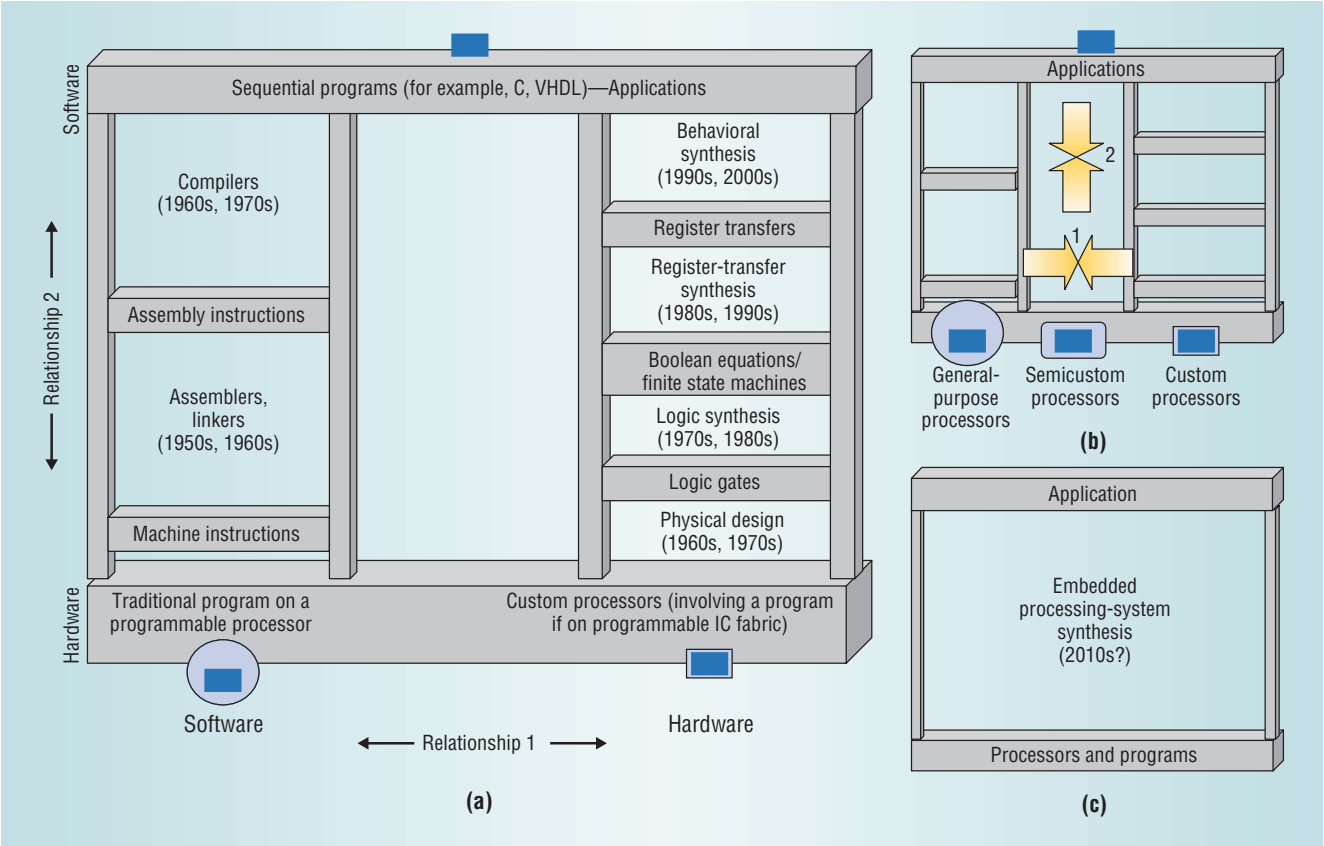

Figure 3. The co-design ladder. (a) Designers use sequential programs to describe complete applications. (b) New tools merge traditional program designs with their coprocessors, and they merge software program designs with their underlying programmable processors. (c) In the future, designers might specify an application and then apply embedded processing-system synthesis.

cache memory size, segment buses, shut down datapath functional units, and scale down supply voltage by configuring the system statically during initialization or dynamically during application execution. Such architectural tuning can reduce power consumption or improve performance.

In short, the processors, memories, and buses—what we previously considered a system's unchangeable hardware—can actually be quite soft.

MERGING DESIGN TOOLS

Increasing hardware programmability is not alone in blurring the distinction between what we traditionally considered software and hardware. Improved hardware design tools are making hardware design look increasingly similar to software design. Designers can use these improved tools to describe their systems' software and hardware in a unified manner.

Climbing the software and hardware design ladders

With the appearance over the past decades of improved design tools, designers can use higher abstraction levels to describe their designs. Using higher abstraction levels is like climbing the rungs of the ladders in Figure 3a, allowing designers to build higher-functionality systems in less time.[8]

Software design tools generally seek to increase the abstraction level at which designers write software. In the 1950s and 1960s, developers used assemblers to program computers using short words, characters, and numbers, rather than bits of zeros and ones.

In the 1960s and 1970s, developers began using compilers that automatically convert sequential programming languages (having "if" and loop statements, subroutines, and so on) to bit-level programs. Continued progress has emphasized even higher abstraction, such as object-oriented programming with associated compilers, which represent rungs even higher than the sequential programs rung in Figure 3a.

Improved hardware design tools have increased the abstraction level for designers too. Developing the hardware ladder rungs took longer, however, for two reasons:

- designing an interconnection of transistors is more complicated than translating sequential programs to programmable processor instructions, and
- implementing part of an application as custom hardware instead of software implies a much stronger demand for optimization.

Nevertheless, in the 1960s and 1970s, developers began using physical design tools to automatically convert transistor circuits to physical chip information. In the 1970s and 1980s, logic synthesis tools emerged for converting state machines and Boolean equations to transistor circuits.

Register-transfer synthesis tools for converting cycle-by-cycle behavior descriptions into Boolean equations and state machines gained popularity in the 1980s and 1990s. Finally, beginning in the 1990s and continuing today, designers can use behavioral synthesis tools that convert sequential programs into cycle-by-cycle behavior.[9]

Thus, designers can now describe a complete application using sequential programming languages (in addition to other computation models such as data flow, hierarchical state machines, and differential equations), independent of whether they eventually implement the application as a custom processor, general-purpose processor, semicustom processor, or collection of processor types.

This ability to describe an application as a unified whole and then automatically implement a collection of programmable and custom processors starkly contrasts with the past. When design processes for different processor types differed radically, developers had to partition applications among programmable and custom processors early in the application development process.

Current synthesis tools still require describing hardware and software applications somewhat differently. Yet the trend is clearly toward eliminating those differences, leading to a second cause of the blurring between software and hardware:

> Today's hardware designers use methods similar to those that software designers use—they write programs and let tools generate the implementation. Thus, not only has the distinction between the physical implementations of traditional software and hardware blurred, but the distinction between the software and hardware design processes, and even between software and hardware designers themselves, is also blurring.

From two ladders to one

The meeting of software and hardware design has ignited the demand for methods and tools to help designers implement applications as collections of processors—generally referred to as *hardware/software codesign*.

Codesign can refer to simultaneously designing the software and the custom processors that make up a system, as relationship 1 in Figure 3a shows. The term can also refer to simultaneously designing a software program and the programmable processor the program will run on, as relationship 2 in Figure 3a shows.

As codesign methods develop and the tools mature, the term codesign will likely begin to lose meaning. The distinction between programmable and custom processors will blur, and the tools to convert applications to processors will merge with one another, as Figure 3b shows.

The distinction between software and hardware design processes, and even between hardware and software designers, is blurring.

The hardware and software ladders might merge into a single design step, in which a designer specifies a desired application and then synthesizes an embedded processing system. The system will consist of a collection of processors and programs, and each program will represent a set of programmable processor instructions or a programmable IC fabric configuration. Several methods and tools are taking us in that direction.

- *Common description methods*. Proposals and standardization efforts such as SystemC (http://www.systemc.org) are focusing on methods to describe complete applications using a single language or environment, rather than using C, C++, or Java for programmable processors and VHDL or Verilog for custom processors.
- *Simulators*. Evolving simulators and debuggers can efficiently simulate complete systems of custom processors, programmable processors executing their instructions, and even analog components.
- *Exploration tools*. Tools to automatically explore the possible tradeoffs of implementing an application as a system with varying numbers, types, and sizes of processors and memories and to automatically generate those implementations are also evolving. Such tools partition applications among programmable and custom processors and schedule the reconfiguration of programmable IC fabrics to implement the custom processors when needed.
- *IC platforms*. Predesigned chips with many programmable and custom processors and memories as well as programmable IC fabrics and their associated compilers, debuggers, and synthesis tools are enabling rapid design of complete embedded processing systems.
- *Semicustom processor tools*. Evolving tools automatically generate the necessary compilers, debuggers, and simulators for user-designed semicustom programmable processors.[4] Likewise, companies are building compiler/synthesis tools to automatically generate

a semicustom processor and a program from a given application.[2]

- *Architecture-aware compilers*. New compilers will be more aware of the underlying programmable processor's resources. Knowledge of cache configuration, DRAM line size, scratchpad memory availability, and functional unit resources can improve the decisions regarding loop unrolling, subroutine inlining, address assignment, instruction scheduling, and shutdown of inactive resources.
- *Dynamic compilation/optimization*. Compilation no longer must be completed before a program executes. Dynamic compilers monitor executing programs, find the most critical regions, and re-optimize them. In fact, hardware itself might have built-in dynamic compiler behavior. For example, some of our work at the University of California, Riverside, focuses on dynamically partitioning by extracting critical regions from a programmable processor's application and dynamically creating a much faster custom processor on a programmable fabric for that region.

As Figure 3c illustrates, future designers might describe the desired functionality of an entire system application using one or more languages. A tool would then implement that functionality. The implementation might consist of the best collection of processors (general-purpose, semicustom, and custom) mapped onto the best collection of IC fabrics (programmable, semicustom, and custom) to meet the design constraints imposed on the system.

The softening of hardware has many implications, perhaps the greatest ones being related to educating the next generation of engineers. Beyond encouraging hardware designers to become better programmers, curriculum designers must fundamentally rethink the introduction of programming and digital design to new engineering and computer science students.

Current engineering and computer science university curricula typically present software and hardware design in separate courses and even as separate tracks. New textbooks for introductory courses will help,[8,10-12] but we need to fundamentally change how we educate embedded systems engineers so they can comfortably cross the traditional hardware/software barrier and build the next generation of application-driven embedded computing systems. ■

Acknowledgment

This work was supported in part by the US National Science Foundation, grants CCR-9876006 and CCR-0203829.

References

1. F.R. Shapiro, "Origin of the Term Software: Evidence from the JSTOR Electronic Journal Archive," *IEEE Annals of the History of Computing*, Apr.-June 2000, p. 69.
2. S. Aditya, B.R. Rau, and V. Kathail, "Automatic Architectural Synthesis of VLIW and EPIC Processors," *IEEE/ACM Int'l Symp. System Synthesis*, IEEE CS Press, 1999, pp. 107-113.
3. J. Fischer, "Customized Instruction Sets for Embedded Processors," *Proc. Design Automation Conf.*, IEEE CS Press, 1999, pp. 253-258.
4. R. Gonzalez, "Xtensa: A Configurable and Extensible Processor," *IEEE Micro*, vol. 20, no. 2, 2000, pp. 60-70; see also www.tensilica.com.
5. N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, Addison-Wesley, 1994.
6. M. Smith, *Application-Specific Integrated Circuits*, Addison-Wesley, 1997.
7. S. Brown and J. Rose, "FPGA and CPLD Architectures: A Tutorial," *IEEE Design & Test of Computers*, vol. 13, no. 2, 1996, pp. 42-57.
8. F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware/Software Introduction*, John Wiley & Sons, 2002; www.wiley.com/college/vahid.
9. D. Gajski et al., *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.
10. W. Wolf, *Computers as Components: Principles of Embedded Computer Systems Design*, Morgan Kaufmann, 2000.
11. F. Vahid, *Digital System Design: A Modern Approach*, John Wiley & Sons, to appear; www.wiley.com/college/vahid.
12. Y. Patt and S. Patel, *Introduction to Computing: From Bits and Gates to C and Beyond*, 2nd ed., McGraw-Hill, 2003.

Frank Vahid *is a professor of computer science and engineering at the University of California, Riverside, and a member of the Center for Embedded Computer Systems, UC Irvine. His research interests include embedded system design methods and architectures. Vahid received a PhD from UC Irvine. He is a member of the IEEE and the ACM. Contact him at vahid@cs.ucr.edu.*