The Softening of Hardware

Unifying Hardware and Software Design in Embedded Systems

Frank Vahid

Dept. of Computer Science and Engineering, University of California, Riverside, and Center for Embedded Computing Systems, UC Irvine

Article accepted for publication in IEEE Computer Magazine

1. Introduction

The first programmable computers were rather nasty beasts, and programming them represented only a small part of the challenge of computing. Those giant machines from the 1940's occupied entire rooms and consumed kilowatts of electricity. The computation problems that engineers faced were mostly related to failed machine hardware components, not to the short and infrequently changed programs being fed to those machines. Engineers viewed computers and their programs as a unified entity.

Over the next decade, the computers became more stable while programs more complex. Solving problems encountered during computing, or "debugging," came to have less to do with removing heat-loving insects stuck in the hot hardware components (which is where the term seems to have originated), and instead 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 – with the first published use of the term "software" apparently being in 1958:

> 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 [8].

The fields of software development and hardware development evolved along separate paths for the latter decades 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. This softening is being impelled largely by embedded systems – those hidden computing systems that drive the electronic products around us, including consumer electronics like digital cameras, personal-digital assistants, music players and video games, office automation equipment like copying machines and printers, medical devices like heart monitors and ventilators, and automotive electronics like cruise-controllers and antilock brakes. Embedded systems squeeze designers through incredibly tight constraints on time-to-market, power consumption, size, performance, flexibility and cost. This squeeze provided the impetus for numerous new technologies over the past two decades that seek to help designers satisfy those constraints.

In this article, we provide a means for understanding these new technologies. We provide a clean distinction between processors, integrated circuit fabrics, and chips themselves. We discuss existing and emerging design tools that help designers build systems of processors using integrated circuit fabrics on chips. We conclude that, once again, we need a unified view of hardware and software. The implications may be greatest on how we educate new engineers.

2. Processors, IC Fabrics and Chips

New technologies tend to earn new names stressing their distinguishing traits, but in the rapidly changing world of computers, such names quickly become outdated. For example, a personal computer that serves files to thousands of users isn't very personal, and today's floppy disks are actually quite rigid. This rapid outdating of names creates confusion. However, at this point in the evolution of embedded computing, we can at least distinguish among three items that are often confused: processors, IC fabrics and chips. We rely on those items to convert our *applications* – those collections of algorithms that comprise the computing behavior of our embedded system products – into reality.

2.1 Processors

A *processor* is a digital design capable of executing an algorithm. A processor typically includes a datapath and controller. The datapath includes basic digital components like registers that store data, functional units that transform data (e.g., adding or shifting), and multiplexors and buses that move data. The controller configures the datapath for such moves, transforms, and stores – a large enough sequence of which we call an algorithm – and is built from basic digital components like registers and logic gates. Notice that we specifically did *not* say that processors are programmable – some processors are customized to one application, while others are programmable to support a variety of applications.

2.1.1 Square Pegs and Round Holes – General versus Custom Processors

"You can't fit a square peg into a round hole" the saying goes. Actually, you can if the hole is big enough.

Embedded system designers must map their applications (pegs) into processors in order to bring those applications closer to life. They can choose from processor types distinguished by their generality versus customizability.

General-purpose programmable processors: The processors most familiar to many of us are general-purpose programmable processors designed to execute nearly any application. Their datapaths consist of large register files and flexible arithmetic-logic functional units. Their controllers have no idea what applications they will run, instead executing memory-stored instructions representing the application. Mapping applications

to such processors utilizes fast (measured in seconds) compilers and other mature tools. These processors are like holes big enough to fit any peg.

Custom processors: In contrast, designers can utilize custom processors to execute an application. Those processors can include just the right numbers, sizes and interconnections of registers and functional units to match the application and to best meet performance and power constraints. They can encode the application right into the controller, eliminating the need for slow and power-hungry accesses to instruction memory. Custom processors may include deep pipelines or large numbers of datapath functional units to achieve tremendous parallelism. Off-the-shelf custom processors may exist for common application computations – such processors are commonly known as coprocessors, accelerators, or peripherals. Otherwise, designers must perform digital design to build their custom processor. Custom processors are like holes cut to fit one peg shape only.

Figure 1 illustrates processor types. Figure 1(a) shows a very simple application involving reading an input stream from a port *p1*, performing a multiply-accumulate using an array of constants *M*, and outputting the final sum over port *p2*. Figure 1(b) represents implementing this application on a general-purpose processor. The statement "$t = t + M[i] * p1$" would likely be implemented using a sequence of steps stored in instruction memory: (1) move *i* from the register file to the data memory address register, (2) read *M[i]* into the register file, (3) store *p1* into the register file, (4) read *M[i]* and *p1* from the register file, multiply them, and store the result back into the register file, (5) read *t* and *M[i]*p1* from the register file, add them, and store the result back into *t* in the register file. In contrast, Figure 1(d) represents a custom processor implementation. The same statement would execute in one step on the custom processor.

Designers like the immediate availability, low-cost (due to mass production), and simple design flow of general-purpose processors. They especially like the flexibility of general-purpose processors – reprogramming can take just minutes, and bugs can be fixed at the last stages of product design. However, they often need the performance, power and cost (in large quantities) advantages of custom-designed processors. Satisfying such competing demands epitomizes the embedded system design problem. In broad terms, the key is finding the balance between general and custom solutions. In specific terms, designers must partition their applications among general and custom processors.

2.1.2 Two Sizes do not Fit All – Semi-Custom Programmable Processors

Not long after processors began finding their way into embedded systems did new classes of processors evolve to satisfy the need for options between the two extremes of general-purpose and custom processors.

Digital signal processors: Embedded computers working in a world of analog signals, like radio, audio, and video signals, typically digitize those signals, transform them, and send new signals out. Unlike the less dynamic data obtained from files or user input that desktop computers operate upon, analog signals tend to stream in and out at rapid rates. Programmable digital signal processors (DSPs) possess special instructions (and supporting underlying hardware) to efficiently handle such streams and common operations applied to them, such as instructions supporting fast reads and writes of large arrays, and instructions supporting single-cycle multiply-accumulate or fast floating-point arithmetic.

Microcontrollers: Other embedded computers operating in a world of events, like buttons being pressed or sensors being triggered, must sense those events and generate new ones in response, like turning on lights or opening doors. These computers have less need for fast memory access and data operations, and more need for bit-level operations, efficient access to external wires, and very low power consumption. Programmable microcontrollers possess architectures tuned to such control behavior, often having narrow datapaths (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 also may closely integrate timers, serial communication, analog-digital converters, and other common embedded control functions.

Semi-custom programmable processors: DSPs and microcontrollers represent early and widespread forms of the growing class of semi-custom programmable processors, also known as application-specific instruction-set processors, or ASIPs [1]. A semi-custom processor is optimized for a particular class of programs, such as programs performing digital picture processing, network routing, or mobile communications. For example, a picture processing ASIP may include datapath components along with special instructions to assist with picture compression.

Figure 1(c) represents a semi-custom processor implementing the earlier example. The earlier statement would be executed 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 *MAC* unit to compute *t+M[i]*p1* and store the result back into *t* in the register file.

The three processor types represent general categories, without boundaries, along a continuum varying from general to custom.

2.2 IC Fabrics

Just as shirts can be made using different materials, so to can processors be built using different IC fabrics. An IC, or integrated circuit, is an interconnection of transistors appearing on a single chip, following one of several possible styles, or fabrics. Some fabrics are customizable to particular components, while others are programmable to support a variety of components.

2.2.1 Once Again, General versus Custom

Transistors represent the fundamental digital entity, which we compose to build the basic digital components in processors, like registers, memories, functional units and random logic units. Just how and when we compose those transistors depends on the IC style, or fabric, in use. Just like processors, those fabrics differ in terms of customizability versus generality.

Custom IC fabrics: We could custom compose those transistors to implement exactly the components in the processor at hand [11]. We could send this complete transistor design to a chip fabrication plant. We thus get a compact, fast, and perhaps even low power implementation. For example, if we need to compute AND and OR functions, we would create a circuit with an AND and OR gate, as shown in Figure 2.

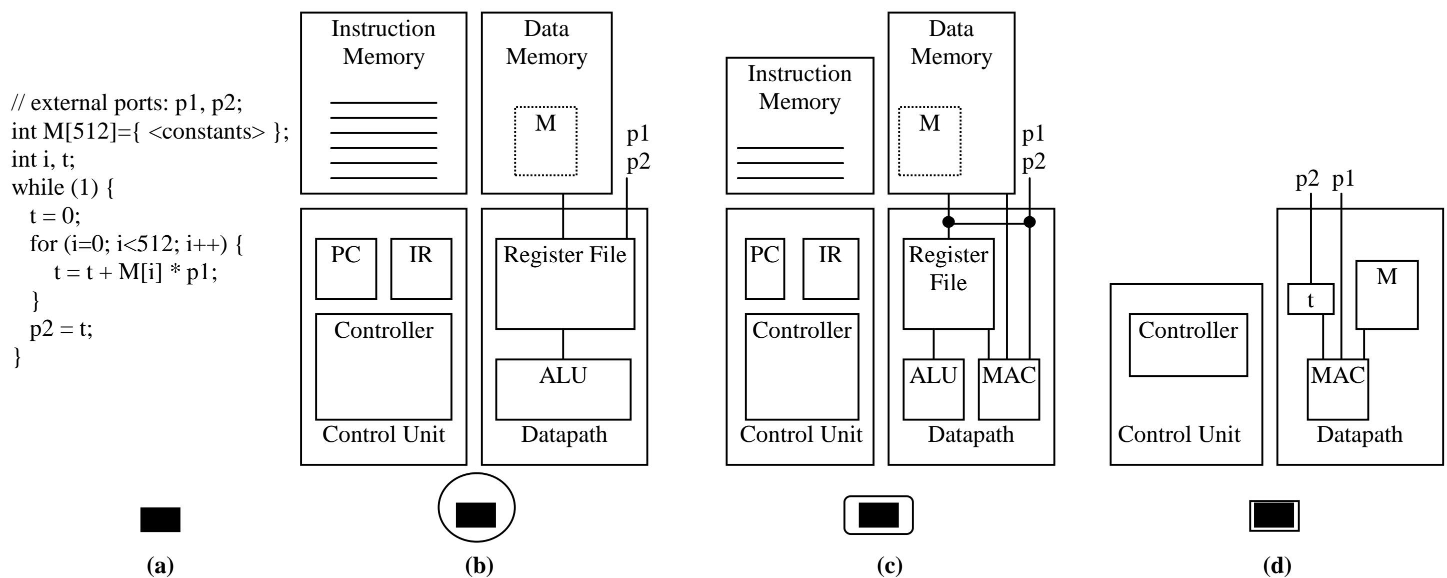


Figure 1: Processors types vary in their customizability to a specific application: (a) a simple application, (b) a general-purpose programmable processor, (c) a semi-custom programmable processor, (d) a custom processor.

Programmable IC fabrics: At the other extreme, a chipmaker could compose transistors into a set of interconnected modules that could each be programmed to implement a large variety of different components [1][9] – thus enabling the chipmaker to pre-fabricate such chips. A module might be, for example, a small memory having four words. By storing the appropriate bits (i.e., a *configuration*) in the memory, a process known as programming[1], we implement our particular desired function. The chipmaker could develop an IC fabric consisting of numerous such combinational modules plus register modules for storage, and could interconnect those modules using programmable interconnect, consisting of multiplexors whose select line is controlled by a bit in another programmable memory. We could thus map any set of digital components onto such a fabric, as long as we had enough resources, since any digital component consists of combinational logic plus storage. Components may be spread out across different modules, and connected components in our processor may exist on modules separated by long distances – but the system will work, albeit consuming much power and having diminished performance. The most popular such fabric today is known as *field-programmable gate array* (FPGA), having that name for reasons we'll discuss later.

The relative 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 and size advantages of custom IC fabrics.

2.2.2 Once Again, Two Sizes do not Fit All – Semi-Custom IC Fabrics

Designers again need more options between custom and programmable IC styles. Several semi-custom styles evolved to meet this need [9].

Standard cell: Custom transistor circuits require much design effort in part to avoid and fix mistakes – place two transistors too close together or make one transistor too small, and the entire circuit may fail. Using pre-designed transistor circuits (libraries) of basic logic components, known as standard cells, reduces such effort. A designer merely places these uniformly sized cells and routes wires to connect them before sending a design to a chip fabrication plant, with fewer errors likely in the fabricated chip than for a custom IC fabric, thus reducing chip design and fabrication time from many months to perhaps just a couple.

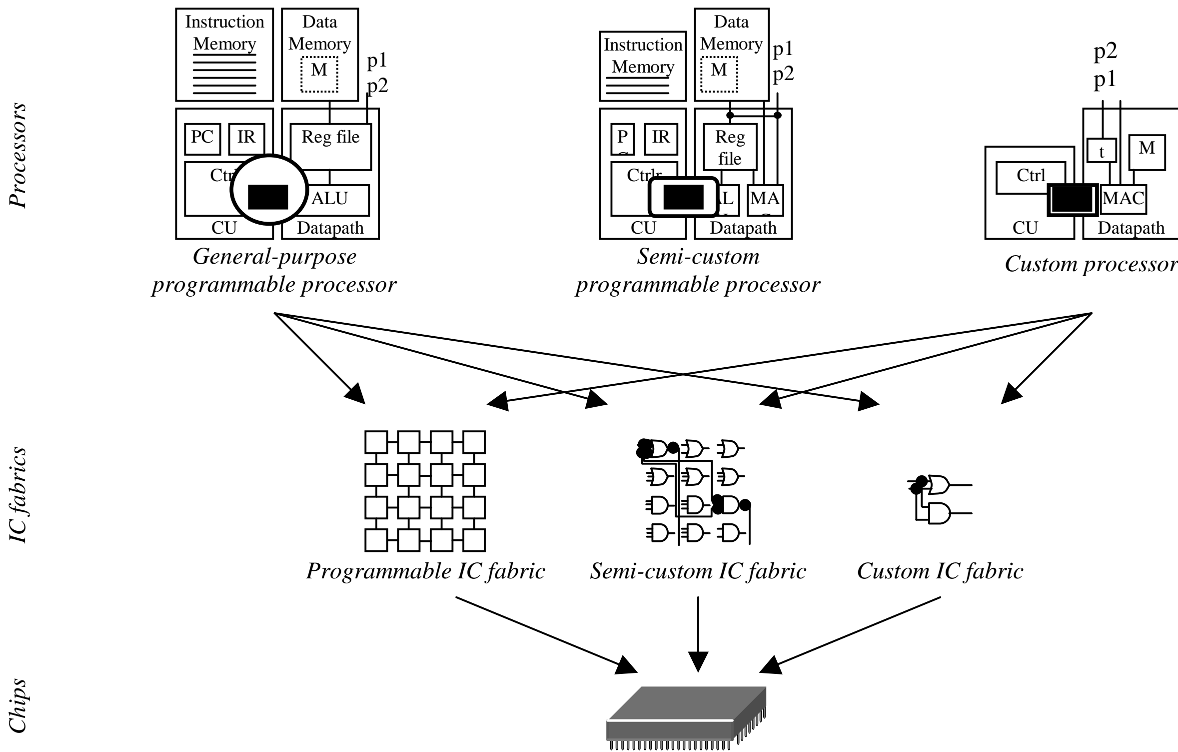
Gate array: A gate array[2] IC fabric reduces chip design time further by pre-placing all the transistors as rows (arrays) of logic gates. The chipmaker can thus pre-manufacture much of the chip. Designers merely need connect the gates, reducing chip design and fabrication time to perhaps just weeks.

---

[1] The term "programming" refers to storing bits in memory, whether for IC fabrics or general-purpose processors. However, the term has also evolved a more common usage referring to earlier steps in such programming of general-purpose processors, namely to the writing of software.

[2] A "field-programmable gate array," or 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 was likely chosen instead due to the feature of pre-placed transistors that is shared with gate array technology, which was popular when FPGAs first appeared.

Figure 2: The relationship among processors, IC fabrics, and chips.



Additional fabrics exist. For example, one fabric is known as *cell array*, in which arrays of cells are pre-placed in rows, so that designers merely need connect them. Another semi-custom fabric involves removing undesired, rather than creating desired, connections, perhaps by using a precise laser. Such a fabric can thus be pre-manufactured, and the destruction of undesired connections, while not today done by a designer in the field, may only take a week or even a day.

2.3 Chips

A chip (also known as an IC) is the thumbnail-sized piece of silicon that physically implements IC fabrics which themselves implement processors. Chips are typically made from silicon, with numerous layers of materials forming transistors and wires that in turn form IC fabrics. Processes for chip fabrication, known as chip or IC technologies, have improved steadily for several decades, with their most prominent improvement being the decreasing *feature size* – roughly defined as the narrowest size of a wire or transistor part. A feature size of around 100 nanometers is becoming common. An IC fabric must appear on a physical chip to enable processors to work in the physical world and thus bring a processor's application to life. When chips were first invented, they integrated tens or hundreds of transistors. Today, they integrate tens or hundreds of *millions* of transistors, due to chip manufacturers taking advantage of decreasing feature size to pack in more transistors. Chip transistor capacity has been doubling roughly every 18 months for several decades, a mind-boggling trend predicted by Intel cofounder Gordon Moore in the 1960's, and known as *Moore's Law*.

High-capacity chips have enabled multiple general-purpose and custom processors to appear on a single chip, thus enabling the high-functionality embedded systems we see today. Several decades ago, a processor might have required numerous chips. In the 1970s and 1980s, we saw the advent of general-purpose processors implemented on one (or just a few) chips, becoming known as *microprocessors*. To distinguish chips implementing custom processors from such microprocessor ICs, the term application-specific integrated circuit, or *ASIC*, grew in popularity. In the 1990s, the term *microprocessor core* became popular to refer to a microprocessor appearing on a chip along with other processors, rather than being on its own chip. The term 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 custom, some semi-custom[3].

High-capacity chips have also enabled multiple IC fabrics to appear on a single chip. Custom and semi-custom fabrics have long co-existed on single chips, but more recently programmable fabrics have also joined in. Furthermore, high capacity chips have enabled the recent technology of

[3] The common method of distinguishing cores as hard, firm or soft has not to do with their processor type, but rather the degree to which they've been pre-implemented; hard cores being almost completely implemented in a physical design in a particular chip technology, firm cores being a structural interconnection of technology-independent gates, and soft cores being a synthesizable behavioral description that requires much additional implementation.

programmable fabrics being implemented on top of semi-custom fabrics – resulting in what are known as FPGA cores.[4]

Today, a complex embedded system with numerous processors may be built using a single chip (known as a system-on-a-chip) or perhaps just a few chips. Notice that each processor may be built from one or more IC fabrics, with those fabrics possibly co-existing on a single chip. A common arrangement is to build a general-purpose processor's datapath using a custom IC fabric, while building the less structured controller from a semi-custom fabric. Programmable fabric may even be used to augment a general-purpose processor with new instructions, leading to an ASIP.

Figure 2 relates processors, IC fabrics and chips. We map processors to IC fabrics, and we map IC fabrics onto chips. Numerous processors often coexist on a single chip and can make use of numerous IC fabrics on the same chip.

3. 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 is understood to represent the soft bits, the zeros and ones, that configure the system to realize a specific application. The "instruction" and "soft bits" usages of the term software are becoming unaligned, as today's designers increasingly use soft bits to represent much more than just instructions.

In particular, today's chips include programmable IC fabrics, which also have memories that must be configured, leading to the following situation that blurs the distinction of software and hardware:

Of the "soft" bits that 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 representing processors often represent custom processors, but may also represent part or all of a semi-custom processor – and can even represent a general-purpose processor.

Two additional trends are worth mentioning here:

- *Reconfigurable computing*: Programmable IC fabrics can be reconfigured even as an application executes, a process known as reconfigurable computing. Processors or parts thereof can be swapped in and out of limited programmable fabric, to carry out the different computations needed by an application at different times.
- *Tunable architectures:* Processors, memories and buses increasingly come with additional configurable features, especially to improve power efficiency. For example, cache memory size can be reduced, buses can be segmented, datapath functional units can be shut down, and supply voltage can be scaled down, through configuration done statically during system initialization or even dynamically during application execution. Such tuning of an architecture can reduce power consumption and/or improve performance.

In short, the processors, memories and buses – what we previously considered the unchangeable hardware part of a system – can actually be quite soft.

4. Design Tools

4.1 Climbing the Codesign Ladder

Increasing hardware programmability is not the only factor blurring the distinction between what we traditionally considered software and hardware. Improved design tools represent another factor that enables designers to describe the software and hardware aspects of their systems in a unified manner. Those tools enable designers to climb to higher abstraction levels, like climbing the rungs of the ladders in Figure 3, thus enabling designers to build higher-functionality systems in less time.

Many design tools in the software domain seek to increase the abstraction level at which designers write software. Soon after computers appeared, assemblers arrived in the 1950s that allowed software developers to work with short words, characters and numbers, rather than with just bits of zeros and ones, when programming a computer. In the late 1950s and 1960s, compilers appeared to automatically convert sequential programming languages – if statements, loop statements, subroutines, and so on – to those bit-level programs. Continued progress has emphasized even higher abstraction, such as object-oriented programming with associated compilers.

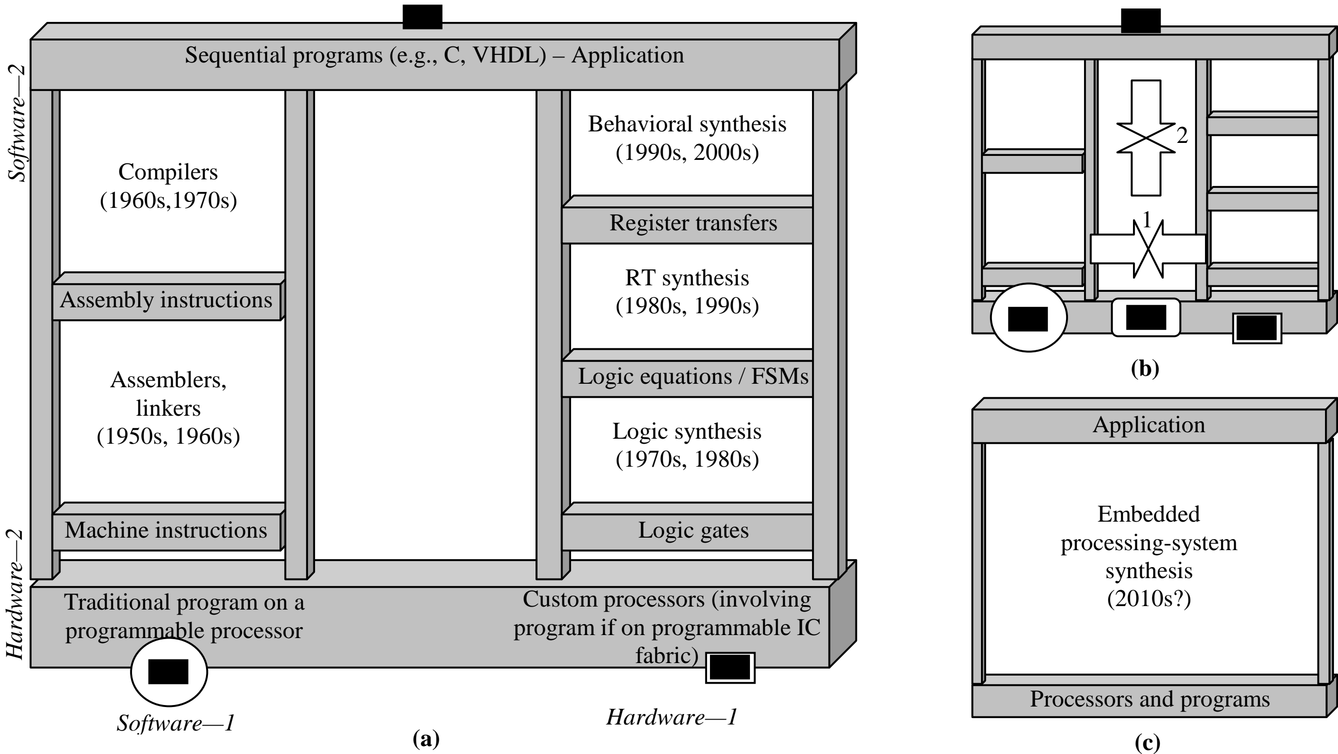
Design tools in the hardware domain also seek to increase the abstraction level for designers, but the development of the ladder rungs took longer than for software. This is partly because designing an interconnection of transistors involves more problems with more degrees of freedom than translating sequential programs to instructions of a programmable processor, and partly because the decision to implement part of an application as custom hardware instead of software implies a much stronger demand for optimization. Nevertheless, the 1960s and 1970s saw the arrival of physical design tools for automatically converting transistor circuits to physical chip information; the 1970s and 1980s saw the arrival of logic synthesis tools for converting state machines and Boolean equations to transistor circuits; the 1980s and 1990s saw the development of register-transfer synthesis tools for converting cycle-by-cycle behavior descriptions into logic equations and state machines; and the 1990s and 2000s saw the arrival of behavioral synthesis tools that convert sequential programs into cycle-by-cycle behavior.

Therefore, designers can now describe a complete application using sequential programming languages[5], independent of whether that application will be implemented as a custom processor, general-purpose processor, semi-custom processor, or some collection thereof. This ability to describe an application as a unified whole and then automatically generate an implementation consisting of a collection of programmable and custom processors starkly contrasts with the past, in which the design processes for different processor types were radically different and thus required a very early partitioning of an

[4] Implementing one technology on top of another is common. For example, one microprocessor can emulate another microprocessor. Programmable IC fabric is usually implemented using IC custom fabric.

[5] Designers can use other computation models too – dataflow, hierarchical state machines, differential equations – for both hardware or software domains.

Figure 3: Design tools are also blurring the hardware/software distinction: (a) the codesign ladder -- design tools have enabled designers to describe a complete application using sequential programs (or something similar), (b) new tools merge (1) the design of traditional programs with their coprocessors and (2) the design of software programs with their underlying programmable processor, (c) eventually these trends may lead to designers specifying an application and then applying embedded-processing-system synthesis to obtain a collection of processors and their accompanying programs, with those programs intended for programmable processors or programmable IC fabrics.



application among programmable and custom processors. Those familiar with the present state of synthesis tools will surely point out that differences still exist in how an application is described for hardware versus software. Yet, the trend is clearly towards eliminating those differences, leading to a second cause of the blurring of the distinction between software and hardware:

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

4.2 Merging the Two Ladders

The coming together of software and hardware design has ignited the demand for methods and tools to assist designers to implement an application as a collection of processors, generally referred to as hardware/software co-design. One type of codesign refers to simultaneously designing a software program and the programmable processor that program will run on – addressing relationship 2 of Figure 3(a) and seen as trend 2 in Figure 3(b). Another type of codesign refers to simultaneously designing the software (to run on a programmable processor) and the custom processors that make up a system – seen as relationship 1 in Figure 3(a) and trend 1 in Figure 3(b). As these trends continue and the tools causing them mature, the term "codesign" will likely begin to lose meaning. Instead, we may see a merging of the two ladders into a single design step, wherein a designer specifies a desired application, and then synthesizes an embedded processing system. That system will consist of a collection of processors and programs – with each program representing either a set of instructions for a programmable processor, or a configuration of a programmable IC fabric. Several tools are taking us in that direction.

Common description methods: Proposals and standardization efforts are appearing that focus on methods for describing complete applications using a single language or environment (e.g., SystemC [6]), rather than using languages like C, C++ or Java for programmable processors and separate languages like VHDL or Verilog for custom processors [7].

Simulators: Simulators and debuggers are evolving that efficiently simulate a complete system consisting of custom processors, programmable processors executing their instructions, and even analog components, before a chip implementation exists.

Exploration tools: Tools are also evolving to automatically explore the tradeoffs obtainable by implementing an application as a system with varying numbers, types and sizes of processors and memories, and to automatically generate those implementations. These include tools that partition applications among programmable and custom processors, and that schedule the reconfiguration of programmable IC fabric to implement those custom processors when needed.

IC platforms, which are pre-designed chips with numerous programmable and custom processors and memories as well

programmable IC fabrics, along with associated compilers, debuggers, and synthesis tools, are appearing to enable rapid design of complete embedded processing systems.

Semi-custom processor tools: Tools are evolving that automatically generate the necessary compilers, debuggers and simulators for a user-designed semi-custom programmable processor [5]. Likewise, combined compiler/synthesis tools are evolving that automatically generate a semi-custom processor plus a program from a given application [1].

Architecture-aware compilers: Compilers are evolving that have a greater awareness of the underlying programmable processor's resources. For example, decisions regarding loop unrolling, subroutine inlining, address assignment and instruction scheduling can be greatly improved through knowledge of cache configuration, DRAM line size, scratchpad memory availability, and functional unit resources.

Dynamic compilation/optimization: Compilation no longer has to be completely done before a program executes. Dynamic compilers monitor an executing program, find the most critical regions, and re-optimize them. In fact, hardware itself may have such dynamic compiler behavior built in.

In summary, not only are hardware and software merging from the perspective of chips themselves, but also the tools and languages used to design hardware and software are themselves merging.

Ultimately, the distinction between hardware and software design may be largely eliminated. As illustrated in Figure 3(c), a designer in the future might describe the desired functionality of an entire system application, using one or more languages. A tool, shown in the figure as an embedded processing-system synthesis tool, would then generate an implementation of that functionality. The implementation could consist of the best collection of processors (general-purpose, semi-custom, custom), mapped onto the best collection of IC fabrics (programmable, semi-custom, custom), to meet the design constraints imposed on the system.

5. Conclusions

Remembering the distinction among processors, IC fabrics and chips can help one understand existing and new embedded system technologies. Trends in these areas as well as in design tools are leading to a blurring of the traditional distinction between hardware and software. Perhaps the most significant conclusion to draw from this merging is the need for greater unification of hardware and software design when we educate embedded system designers – university curricula instead typically present software design and hardware design in separate courses and even as separate tracks. New textbooks for introductory courses may help, but a much more fundamental change is likely needed to achieve the goal of educating embedded systems engineers, so that they can comfortably cross the traditional hardware/software barrier, in order to effectively build the next generation of embedded computing systems.

References

[1] S. Aditya, B.R. Rau, V. Kathail. Automatic Architectural Synthesis of VLIW and EPIC Processors, IEEE/ACM International Symposium on System Synthesis, pp. 107-113, 1999.

[2] Brown, S. and J. Rose. FPGA and CPLD Architectures: A Tutorial. IEEE Design and Test of Computers, 1996.

[3] Fischer, J. Customized Instruction Sets for Embedded Processors. Design Automation Conference, pp. 253-258, 1999.

[4] Gajski, D., N. Dutt, A. Wu and S. Lin. High-Level Synthesis: Introduction to Chip and System Design. Kluwer Academic Publishers, 1992.

[5] Gonzalez, R. Xtensa: A Configurable and Extensible Processor. IEEE Micro, pp. 60-70, 2000. See also http://www.tensilica.com.

[6] Open SystemC Initiative, http://www.systemc.org.

[7] Ramanathan, D., R. Roth and R. Gupta, Interfacing Hardware and Software using C++ Class Libraries. International Conference on Computer Design, pp. 445-450, 2000.

[8] Shapiro, F.R. Origin of the Term Software: Evidence from the JSTOR electronic journal archive. IEEE Annals of the History of Computing, 22(April-June):69, 2000. Refers to an article by John W. Tukey in the January 1958 American Mathematical Monthly, titled "The Teaching of Concrete Mathematics."

[9] Smith, M. Application-Specific Integrated Circuits. Addison-Wesley, 1997.

[10] Van Meerbergen, J., A. Timmer, J. Leijten, F. Harmsze, M. Strik. Experiences with System Level Design for Consumer ICs, VLSI'98, pp. 17-22, 1998.

[11] Weste, N. and K. Eshraghian. Principles of CMOS VLSI Design. Addison-Wesley, 1994.