September 2007

EMBEDDED COMPUTING

It’s Time to Stop Calling Circuits “Hardware”

Frank Vahid
University of California, Riverside

Expanding the software concept to spatial models like circuits will facilitate programming next-generation embedded systems.

In computer technology, names often change. Even the word "computer" previously referred to a human whose job was to crunch numbers, but today refers to a machine. The advent of field-programmable gate arrays requires that we stop calling circuits "hardware" and, more generally, that we broaden our concept of what constitutes "software." Doing so will help establish a cadre of engineers capable of maximally exploiting modern computing platforms that involve multiprocessors and FPGAs.

When the term software was coined in the late 1950s to raise awareness that writing programs was becoming as important as the design and maintenance of the hardware that then dominated computing efforts, the meanings of software and hardware were clear. Hardware referred to physical components like boards, fans, tubes, transistors, and connections—and could be touched. Software referred to the bits representing a programmable processor's program. During the decades that followed, building programs (software design) evolved to require dramatically different skills than connecting physical components into circuits (hardware design).

IMPLEMENTING CIRCUITS

Today, embedded-system designers frequently supplement microprocessors with custom digital circuits, often called coprocessors or accelerators, to meet performance demands. A circuit is simply a connection of components, perhaps low-level components like logic gates, or higher-level components like controllers, arithmetic logic units, encoding engines, or even processors.

FPGAs as software

Increasingly, designers implement those circuits on an FPGA, a prefabricated chip that they can configure to implement a particular circuit merely by downloading a particular sequence of bits. Therefore, a circuit implemented on an FPGA is literally software.

The key to an FPGA's ability to implement a circuit as software is that an *N*-address-input memory can implement any *N*-input combinational circuit. For example, given the function $A'B'C' + ABC$, it's possible to simply connect *A*, *B*, and *C* to the address inputs of a three-address-input (eight-word) memory, store a 1 into the first and last memory words, and store 0s into the other six words. The memory will then output 1 when $ABC = 000$ ($A'B'C'$) or when $ABC = 111$ (ABC), and will output 0 for any other values of *ABC*.

An FPGA has thousands of small memories each having perhaps four inputs and two outputs, and each known as a lookup table. A circuit with more than four inputs can be partitioned into four-input subcircuits, each mapped to a LUT. To support sequential circuits, LUT outputs feed into flip-flops, collectively known as a configurable logic block. A multiplexor, along with a programmable register that controls the multiplexor's select lines, chooses whether the CLB output comes directly from the LUT or from the flip-flop.

Finally, CLBs are preconnected through miniature crossbar-like switches, in which programmable registers also determine the actual paths, to ultimately achieve a particular circuit's connections. A developer thus can implement a desired circuit on an FPGA by writing the proper bits to each LUT and programmable register, as determined by design automation tools.

The fact that developers can implement circuits as software on FPGAs is a very big deal because circuits can implement some computations hundreds or even thousands of times faster than microprocessors. The speedups come from a circuit's concurrency at the bit level, arithmetic level, or even process level, because every circuit component executes simultaneously. For example, reversing the bits of a 32-bit number, performing 50 independent multiplications, or executing 20 processes each requires a long sequence of instructions on a microprocessor. But on an FPGA, the bit reversal is just crossed connections, the multiplications could execute concurrently on 50 multipliers, and the processes could execute in parallel as 20 different circuits.

Underutilizing FPGAs

Computationally intensive applications involving video or audio processing, compression, encryption, real-time control, or myriad other tasks found in domains like consumer electronics, communications, networking, industrial automation, automotive electronics, medical devices, office equipment, and much more could thus execute hundreds or even thousands of times faster as circuits on FPGAs than as instructions on microprocessors—without the rigidity, high cost, and long turnaround times associated with building new chips. Such speedups, achieved using the changeable medium of software (on FPGAs), enable new applications not otherwise possible.

Yet, embedded-system application designers working with software often overlook FPGAs, even when FPGA performance gains would tremendously benefit an application while satisfying cost and power constraints. And designers never even conceive many applications that FPGA performance would enable because they don't consider that software can meet those applications' performance demands. However, because an embedded-system designer faces the challenge of making design tradeoffs under multiple excruciating constraints on design metrics like performance, power, cost, size, and design time, the effective designer can't be strong with just microprocessors or FPGAs; these designers must be capable of finding just the right balance of both.

Part of the problem is that computing-oriented application designers tend to be unfamiliar with FPGAs, which in turn occurs largely because the engineering design and education communities continue to treat circuits as hardware. Until the advent of FPGAs, treating circuits as hardware made sense. Designers implemented circuits either by connecting physical components or by creating a new physical chip—things they could touch.

Yet today, even with circuits increasingly implemented as software on an FPGA, circuit designers are still known within companies as hardware people, circuit-design languages are still called hardware description languages, algorithms ported to circuits are still known as hardware algorithms, and coprocessing circuits are still known as the hardware in hardware/software systems. But with the FPGA market generating billions in annual revenue from the sale of hundreds of millions of devices to thousands of customers, the categorizing of circuits as hardware is grossly inaccurate.

Effect on computing field

That circuits on FPGAs are actually software rather than hardware isn't just an insignificant technicality, like a tomato actually being a fruit rather than a vegetable. On the contrary, treating circuits as hardware hurts the computing field, especially embedded systems. The reason is that most computing-oriented engineers—software designers—aren't interested in learning about hardware design. Hardware to them is a different beast. Thinking that circuits are hardware, computing-oriented engineers arenmotivated to learn to design for FPGAs, and perhaps even more significantly, computing-oriented college students often avoid courses that involve circuits and FPGAs, due to not being "hardware types."

Temporal versus spatial modeling

A typical computer science department will have computing-oriented students with programming specializations in graphics, the Web, or
real-time system development, for example. But rarely do computer science students specialize in FPGA development; those students are typically in electrical engineering departments, where "hardware" is the focus.

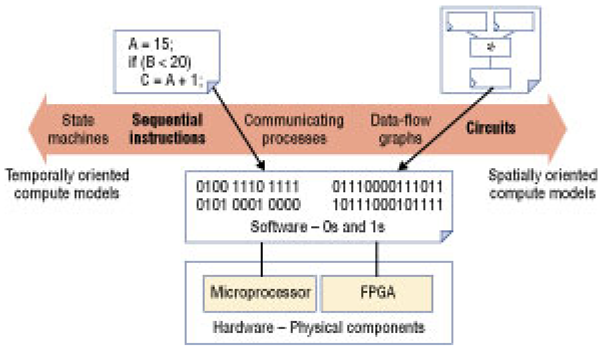

Figure 1. Circuits as software. Temporally and spatially oriented computing models can both be compiled to software that is downloaded into hardware components.

The issue actually extends well beyond circuits and hardware and into the realm of broadening the concept of computation models to consider both temporally and spatially oriented models, a concept illustrated in Figure 1. Developing modern embedded software capable of executing on multiprocessor and FPGA platforms requires expertise not just in temporally oriented modeling (*W* comes after *X*) like writing sequential code but also in spatially oriented modeling (*Y* connects with *Z*) like creating circuits.

The former involves ordering tasks in time, while in the latter, tasks execute concurrently in a parallel or pipelined fashion. Sequential coding languages like C excel at capturing temporally oriented computations. Designers traditionally capture spatially oriented computations in hardware description languages, but they also capture them using extensions to sequential languages, such as Posix threads or even SystemC's explicit circuit extensions to C++. In fact, sequential code coupled with knowledge of how a synthesis tool will convert that code to a circuit can capture many spatially oriented computations. Circuit designers do this all the time when they write loops that they know will be unrolled, for example.

Software developers tend to excel at the temporally oriented task of creating sequential code models, like defining algorithms, subroutines, and so on. But they're typically weaker at creating models that also involve some amount of spatial orientation, like parallel processes, data-flow graphs, or circuits, largely because computing education in universities tends to emphasize the former with little attention given to the latter. Yet with embedded systems continuing to grow in importance, such imbalance can't persist much longer.

Introducing students to circuits

What can be done? For starters, computing departments in universities might reconsider how they introduce circuits to their students. Most computing curricula today introduce circuits as the foundation of how microprocessors work, and perhaps as glue logic for interfacing physical components. But the modern computing curriculum should also introduce circuits early on as a model for describing computations, representing a highly parallel alternative to a sequential code model.

For example, a temporally oriented approach to sorting *N* data items makes several sequential passes over the data, swapping certain data items along the way. A spatially oriented approach might use *N*/2 components to order pairs of data, followed by *N*/4 components to order quadruples of data, and so on in a pipelined manner. The ways of thinking of those two computation approaches differ dramatically. With the advent of multiprocessors and FPGAs, who's to say that the spatially oriented way is any less important?

Developing students' spatial thinking via computing-oriented circuit design, when coupled with the existing developing of their temporal thinking via traditional sequential programming, could lead to students more easily learning parallel-programming concepts, something many writers in recent years insist must be strengthened. Parallel-programming concepts are really a hybrid of temporal and spatial thinking. Building both temporal and spatial skills before embarking on parallel programming can be thought of as building both the left and right wings of a plane before attempting to fly.

Meanwhile, along with changing university curricula, we all (whether in universities or companies) can stop calling circuits hardware, and we also can broaden our use of the word software beyond just microprocessors. In this way, circuits and other modeling approaches with a spatial emphasis can assume their proper role in the increasingly concurrent computation world of software development.

Frank Vahid is a professor in the Department of Computer Science and Engineering at University of California, Riverside. Contact him at vahid@cs.ucr.edu.

Editor: Wayne Wolf, School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta; wayne.wolf@ece.gatech.edu