GUEST VIEWPOINT: EMBEDDED SYSTEMS AND THE MICROPROCESSOR

Beginnings of the Long Downhill for the Microprocessor

By Nick Tredennick & Brion Shimamoto {4/24/00-02}

The ascent of microprocessor-based applications emphasized the efficiency of the designer (e.g., high-level languages, compilers, operating systems, library functions) over the efficiency of the design (e.g., time, size, power, performance). The rapidly increasing market in compute-intensive portable consumer applications will require a design method that emphasizes the efficiency of the design. This is the story of the design methods engineers use to solve problems: the history of design methods, how they influence (and are influenced by) real-world applications, and where we think they are headed in the near future. It is the story of how our 50-year romance with the computer and with computer architecture has us on a course that may now *retard* our ability to solve emerging problems. Here's why.

The Way It Is

Computer scientists emphasize the role of microprocessors as the CPU in a computer system. However, as the number of personal computers shipped each year exceeds 140 million units worldwide, the number of embedded microprocessors shipped may exceed 5 billion units worldwide. That means that microprocessors used as a CPU are two percent of the total volume of microprocessors. The market for embedded microprocessors has long dominated unit volumes, but it has been a low-margin business. That's changing with increasing demand for portable electronic devices and with increasing computational requirements in the applications. The market for embedded applications will drive the electronics industry in the future, and it will take the designers with it.

First, a short history of electronic problem solving.

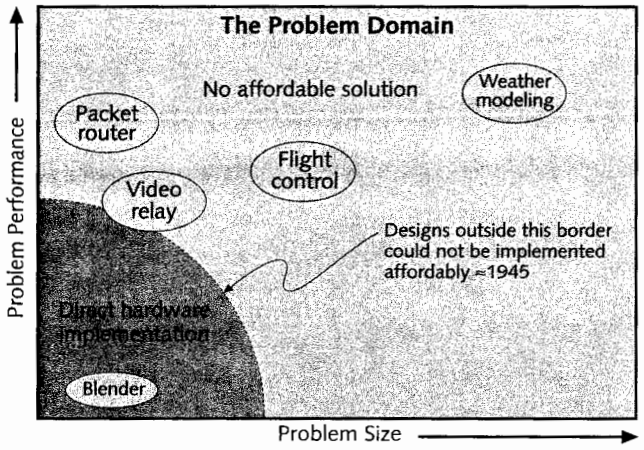

Figure 1. Some problems are larger than can be affordably solved using a direct hardware implementation.

Problem Solving

The solution to a problem is an algorithm using a set of hardware resources.

An engineer asks, "How do I solve this problem using available hardware resources?"

Let's look at the domain of problems and at the range of solution types.

Problem Domains and Solution Types

Before the computer, engineers solved problems by mapping an algorithm directly into hardware. That is, the resulting hardware reflected the solution directly. One could infer the algorithm from studying the hardware components. So the resulting solution was a fixed algorithm on a fixed set of resources. We call this *fixed resources and fixed algorithms.*

The problem domain looks something like Figure 1, with the problem size on the horizontal axis and with the required performance on the vertical axis. In 1945, before the computer, engineers solved problems by direct hardware implementation. The affordability of hardware limited the range of problem sizes and the range of performance needs that could be tackled. It wasn't possible in 1945, for example, to build affordable packet routers or electronic flight-control systems or to solve the real-time weather-modeling problem. As underlying hardware components improved, the range of affordable, direct hardware solutions expanded.

The Computer—a New Way

The computer was a breakthrough in problem solving. Here was a way to implement any algorithm using a fixed set of resources. The computer provided fixed resources in the form of ALUs, shifters, floating-point units, registers, and the like. The computer could trade time for resources, in that an algorithm on the computer could iterate to solve a problem. The computer solved problems—ones that would have required unaffordable hardware resources—by providing limited (affordable), fixed, hardware resources and by sharing those resources over time.

In addition, the same hardware could solve multiple problems. Thus, the computer hardware could be amortized both over time and over a range of problems. The computer introduced a new type of solution—one with *fixed resources and dynamic algorithms.*

Figure 2 shows the effect of the computer on the problem domain. The computer could not solve every problem in the domain. It could not, for example, meet the performance needs of problems such as real-time weather forecasting. For small problems, the direct hardware approach was still cheaper. The computer could, however, solve a problem of any size—if you could afford to wait for the answer.

Direct hardware implementation carved out an area of the problem domain. Problems outside that area could not be solved economically with a direct hardware implementation. The area staked out by direct hardware implementations, however, grew as hardware components improved. With its introduction, the computer staked out its problem domain: large problems with modest performance requirements. As hardware components improved, the range of computer-based solutions grew. When the computer was introduced, the areas of the problem domain occupied by applications for the computer and by direct hardware implementation did not overlap: problems small enough to be solved economically by direct hardware implementation were too small to be economically solved on a computer.

Since the algorithms are variable, the computer amortizes the cost of its fixed resources across a range of different problems, or over time for a single large problem. Problem solving became *programming*—the mapping of algorithms onto a computer's fixed hardware resources.

Competing Problem-Solving Methods

The computer introduced the engineer to a new way of solving problems. An engineer could employ either a direct solution or a computer-based solution. For the direct solution, the engineer selected hardware resources and mapped the algorithm directly into hardware by designing the hardware resources (datapaths) and their controller (state sequencer). For the computer-based solution, the engineer mapped (i.e., *programmed* or sequenced) an algorithm onto a computer's (fixed) resources. The computer provided the hardware resources and state sequencer, and it provided an instruction set to enable the user to map algorithms onto the hardware resources. The engineer no longer had to select the appropriate hardware resources and also no longer had to design the state sequencer. Problem solving became programming.

The two problem-solving methods have been in competition with each other. In the problem space, there is an equal-cost boundary between the two methods, as Figure 3 shows. Over time, the boundary moves as the methods' underlying components improve.

If the application requires the ultimate in performance, a direct hardware solution is indicated: *fixed resources and fixed algorithms.*

But if the problem is too expensive to solve with a direct hardware solution, a computer-based solution is indicated: *fixed resources and dynamic algorithms.*

Direct hardware solutions advanced with the introduction of the transistor (announced by Bell Labs in June

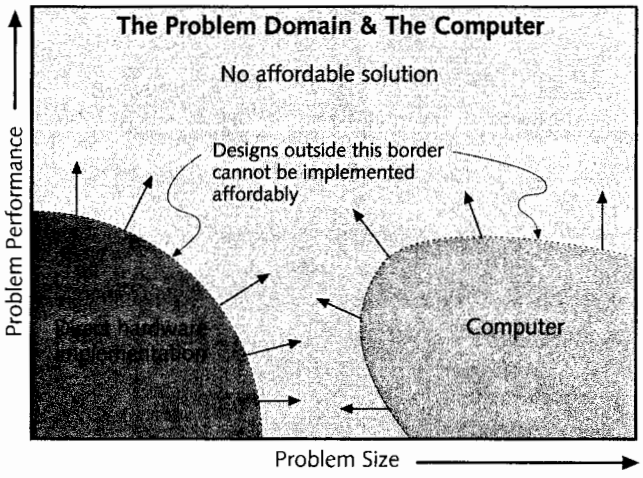

Figure 2. The computer solved problems that had been too expensive to solve.

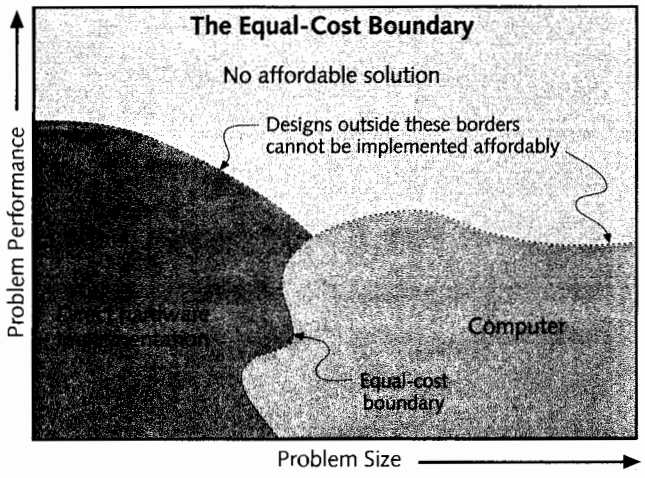

Figure 3. The equal-cost boundary between direct hardware implementation and computer-based solutions.

1948), but so did computer-based solutions as both types of solutions became smaller, cheaper, and more reliable.

Direct hardware solutions advanced shortly after the invention of the integrated circuit with the introduction of digital "logic families" (see Glushkov [1965], Karp & Miller [1969], and Claire [1973]). After a brief experimenting and competitive period, the TTL (transistor-transistor logic) logic family dominated. TTL manufacturers offered designers a range of electrically compatible digital logic-macro functions. TTL raised the efficiency of the engineer who was designing and building direct hardware implementations. Engineers worked with building-block logic-macro functions rather than spending time building low-level functions. TTL expanded the range of problems amenable to direct hardware implementation, and it soon dominated much of the problem domain.

Introduction of the Microprocessor

The TTL family of logic-macro functions and its derivatives grew in scope and in complexity until it included ALU functions and bit-slice components. The growing complexity of logic-macro functions led to commercial development of the microprocessor. The microprocessor became commercially available in 1971.

As Figure 4 shows, with the advent of the microprocessor, the area of the problem domain occupied by direct hardware implementation began to give way to microprocessor-based solutions. Building controllers and mapping algorithms into direct hardware implementations lost ground to the practice of programming algorithms onto the microprocessor's fixed resources.

The embedded microprocessor brought computer-based problem-solving methods to areas of the problem domain that had been dominated by logic-macro-function solutions. (Originally, "embedded microprocessor" would have been redundant.) Improvements in semiconductor fabrication expanded the range of affordable applications for embedded microprocessors, displacing logic-macro-function solutions. As the microprocessor got faster and more capable, it was able to solve larger problems more affordably, and it was able to solve problems needing more performance. At the same time, improvements in semiconductor fabrication decreased the size of older microprocessors, making them cheaper and driving them into applications where they were once uneconomical.

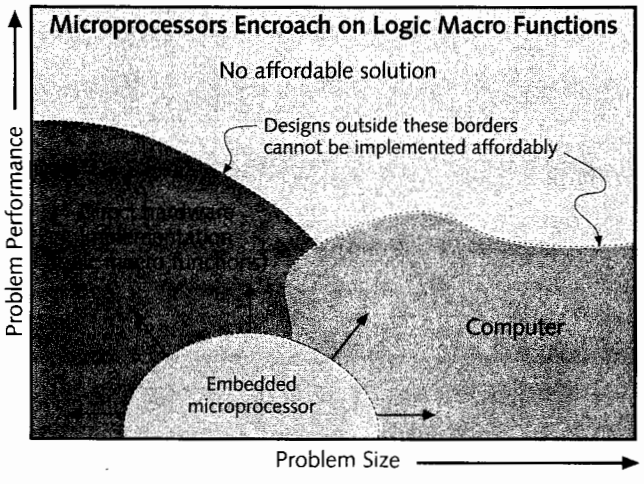

Figure 4. The microprocessor encroaches on problem domains belonging to logic-macro functions.

Abstraction and Design Efficiency

Moving from discrete components to logic-macro functions raised the designer's level of abstraction. The engineer solved problems with ALUs, registers, and multiplexers rather than with resistors, capacitors, and transistors. Moving from logic-macro functions to microprocessor-based design (solving problems by programming) raised the designer's level of abstraction again. The problem changed from building a direct hardware implementation with logic-macro functions to programming the microprocessor's fixed resources. Abstraction improves design in two ways:

- The engineer's efficiency improves as building blocks encompass more low-level functions. Families of compatible logic-macro functions, with their fixed range of input and output interfaces (supply voltage, fan-out, fan-in, minimum and maximum logic levels, and other parameters), free the designer from concern about these low-level functions. Designing with families of logic-macro functions isn't a perfect solution, because as the engineer moves a step closer to solving the problem at hand, there is an accompanying degradation in efficiency. The solution degrades because some logic-macro functions use more power than they need (some transistors driving only one load are large enough to drive 10 loads), and they cannot be used with complete efficiency (the circuit may use only three of eight units in a packaged component).
- The cost of components decreases. Applying the same building blocks to a range of problems increases production volumes (by amortizing the component design cost across a larger base).

The designer's efficiency rises and the cost of components falls as the level of abstraction increases. The efficiency of the design decreases as the level of abstraction rises, however. The efficiency of a microprocessor-based design suffers because, for example, mapping the programming language into the microprocessor's instruction set isn't perfect, and because the microprocessor's fixed resources are unlikely to be a perfect match for the requirements of the application.

Embedded microprocessors compete with logic-macro functions in the problem domain. A logic-macro-function-based solution is more direct, but it uses a greater diversity of components. The embedded microprocessor is less direct, but it uses a few standard components (microprocessor, ROM, RAM, and peripheral chips) for a wide range of applications and thereby achieves enormous volumes and low

cost. Logic-macro-function solutions prevail in areas of the problem domain where an embedded microprocessor cannot meet the performance requirements, and in areas where the problem is too small to be solved economically with an embedded microprocessor.

Logic-macro functions displaced discrete components because they raised the efficiency of the designer, and because applications tend to be cost driven and not performance driven. (The lower-cost logic-macro-function-based solution was cheaper and had adequate performance.) Now the microprocessor is displacing logic-macro functions for exactly the same reasons (lower cost and adequate performance). Growing demand for microprocessors encourages many semiconductor companies to design and to manufacture microprocessors.

Forty Years of Computer Education

When computers were huge, computer design courses were taught as an extension of logic design. The first commercially available microprocessors changed that.

The microprocessor was designed to displace logic-macro functions or ASICs (application-specific integrated circuits) in direct hardware implementations (see Figure 4), and in most circumstances it has. As semiconductor fabrication improved, the microprocessor's performance grew to intersect the range of computer applications. These were still the days when computers were not microprocessor based. In 1974, the microprocessor was offered as the CPU in a hobbyist's computer system. Within 10 years, workstations and personal computers based on microprocessors were flooding the market.

The popularity and accessibility of microprocessors and of computer systems fed the enthusiasm for "computer architecture" in industry and at universities. The design of instruction sets and of microprocessors became popular by the early 1980s.

At universities, computer design courses went from emphasizing logic design for large machines and minicomputers to emphasizing the design and implementation of instruction sets for microprocessors.

Fads Drive Industrial and Academic Research

As denizens of high-tech, we may fancy ourselves akin to *Star Trek*'s Mr. Data, but we are just as driven by fads as the fashion and toy industries. We see fads, for example, in generations of computer and microprocessor design.

Electrical engineers designed the first microprocessors for embedded applications in the problem domain. Lee Boysel's AL1 was probably the first microprocessor to be embedded in a commercial product, appearing in a data terminal that Four-Phase Systems shipped in 1969. Intel's 4004—introduced in 1971 and credited as the first commercially available microprocessor—was originally designed to replace the custom chips (ASICs) in a calculator design. "Computer architects" who designed microprocessors for CPU applications followed the microprocessor's pioneers.

The first wave of microprocessor computer architects (this is the second wave of microprocessor designers and the n^{th} wave of computer designers) tried to design instruction sets matched to a target computer language. These microprocessors were later dubbed "CISCs" (complex instruction-set computers).

For example, according to *The 8086 Family User's Manual*, "Software for high-performance 8086 and 8088 systems need not be written in assembly language. The CPUs are designed to provide direct hardware support for programs written in high-level languages such as Intel's PL/M-86."

The second wave of microprocessor computer architects tried to design instruction sets that would be easy targets for compilers. These microprocessors were called RISCs (reduced instruction-set computers). The RISC fad began in the early 1980s, calling for simplifying the instruction set to make the processor faster.

SPARC, a second-wave microprocessor, began as a research project at U.C. Berkeley and was later adopted by Sun Microsystems. It is also a good example of a second-wave microprocessor; according to the article by Dave Ditzel, "Why RISC Has Won," in *The SPARC Technical Papers*, "CISC systems had tried to convey power through the instruction set alone, but all too often, these optimizations merely ended up crippling the hardware and compiler. ... RISC takes the opposite approach. RISC instructions make possible the creation of simple, high-performance hardware implementations. RISC instructions are simple to decode, are fixed in length, are simple to pipeline and have few side effects. These simple instructions give the compiler more opportunity to apply optimizations that improve the quality of the code."

The current generation of microprocessor computer architects is trying to design instruction sets that exploit parallelism in the instruction stream. These microprocessors are called VLIW (very long instruction word) or EPIC (explicitly parallel instruction-set computing).

Intel's next-generation microprocessor, the IA-64, exemplifies the current generation. According to Linley Gwennap and Kevin Krewell in their book *Intel Microprocessor Forecast 1H00*, "EPIC relies on the compiler to create groups of instructions for parallel execution (hence the 'explicitly parallel'). By moving the complexity of instruction scheduling from the hardware into the compiler, this technique should reduce the amount of hardware needed to achieve high performance and should decrease pipeline length by eliminating the stages needed to schedule instructions. Given a good compiler, an EPIC processor should achieve better performance than a RISC processor with a similar transistor budget. Of course, any move away from the awkwardness of the current x86 architecture (IA-32, in this nomenclature) would be a positive step."

Computer Architecture is Folklore

Despite its trappings and status, the field of computer architecture is weak. It is driven by fads and it is based on folklore.

Application of the scientific method is rare and there is no rigor in experimentation. Each wave of computer architects believes that the last generation knew nothing and that the current generation has the answers. In this way, the computer architecture community is like a perennial teenager: It never grows up. Here's an example from a paper by Charles Church that is as old as the microprocessor. Entitled "Computer instruction repertoire—Time for a change," the paper was presented at the Spring Joint Computer Conference in 1970: "There are two major areas that must be attacked if a job-oriented computer is ever to exist: a) excessive editing, b) fixed-length instructions .... Excessive editing largely results from two basic and common weaknesses in modern computers: a) inflexible word or character addressing, b) excessive register orientation."

Charles Church's paper presents an analytical case and concludes that to make progress in instruction-set design, computers must move away from fixed-length instructions, inflexible word addressing, and excessive register orientation. These will be key among many, many principles at the heart of the RISC fad more than 10 years later. Dave Ditzel, in "Why RISC Has Won," expressed the arrogance and immaturity of this (typical) generation of computer architects well: "A Reduced Instruction Set Computer (RISC) is really about good engineering design tradeoffs."

As if previous generations of engineers had not been able to make good design tradeoffs.

The computer industry is growing so fast that it isn't forced to grow up. Everything we try works. Fundamental assumptions change with each generation as we learn more about compilers, languages, decoders, pipelining, and caches, for example. Improvements in semiconductor fabrication mask (no pun intended) our antics with instruction sets. Substantive improvements in one area mask the folly in another. Despite insistence that there's rigor in the analysis, the field has a history of opinion based on popular fads.

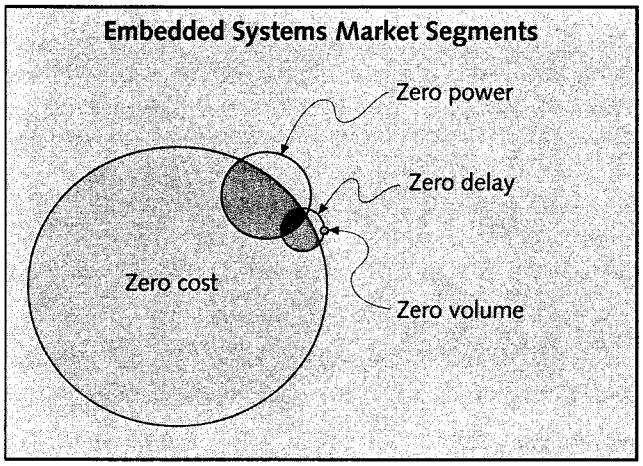

Figure 5. Segments in the embedded systems market.

What's Next? The Embedded Systems Market

The embedded systems market, which includes applications for embedded microprocessors, includes the high-volume consumer applications that drive the electronics industry. Analysis of the embedded-systems market, which is the market for all but 2% of microprocessors, will tell us where the electronics industry is headed. The embedded systems market consists of four overlapping segments, defined by their design requirements. The dominant-characteristic taxonomy of the embedded systems market is illustrated in Figure 5.

Zero Cost

The *zero-cost segment,* which, to a first approximation, represents almost all of the embedded systems market, is the segment for which low cost is the overriding consideration. Most microprocessors go into consumer appliances (microwave ovens, electric razors, blenders, toasters, and washing machines) that generally have minimal processing needs. These are commodity markets: that means they sell in high volumes (millions of units to tens of millions of units). These markets are characterized by intense price competition, so substantial effort goes into reducing production cost. The ideal would be zero cost to implement.

Zero Power

The *zero-power segment,* which, to a first approximation, represents a small percentage of the embedded systems market, is the segment for which zero-power dissipation represents the ideal. These applications are consumer items, such as smoke detectors, cellular phones, pagers, pacemakers, hearing aids, MP3 players, and pocket calculators, which should run forever on a single button-size battery or on weak ambient light. As with all consumer applications, minimum product cost remains a concern.

Zero Delay

The *zero-delay segment,* which, to a first approximation, represents a little more than zero percent of the embedded-systems market, is the segment for which zero delay from data in to result out represents the ideal. These applications are also consumer items, such as high-end printers, scanners, copiers, and fax machines, for which processing power and throughput are important—at minimum product cost, of course.

Zero Volume

The *zero-volume segment,* which, to more than a first approximation, represents zero percent of the embedded-systems market, is the segment for which the application potential is nearly zero. If the application volume is going to be very close to zero, there must be some other reason to attempt to capture the application. One motive is public relations. Intel invested considerable money and effort in the design of the 80960MX processor, for which, at the time of implementation, the only known application was the YF-22 aircraft. When the only prototype of the YF-22 crashed, the

application volume for the 960MX actually went to zero; but even if the program had been successful, Intel could not have expected to sell more than a few thousand processors for that application. Intel must have made the investment in the 960MX for reasons other than potential application volume and eventual profit.

Public relations and a leading-edge image motivate support for the zero-volume segment. GM's and Ford's NASCAR racing teams support the auto industry's zero-volume segment (for the same reason Intel once supported the zero-volume microprocessor segment). NASCAR.com serves more pages to sailors in the U.S. fleet than any other Web site.

In the embedded systems market, the zero-volume segment overlaps with the zero-delay segment, but is completely disjunct from the zero-cost segment.

The Leading-Edge Wedge

The taxonomy of embedded applications is important because it indicates where the electronics industry is headed. Continued improvement in semiconductor fabrication, continued proliferation of cellular telephones, and growing popularity of handheld devices (digital cameras, GPS receivers, PDAs, etc.) drive more computing into portable devices. Because they are consumer devices, they fall into the zero-cost segment. Because they have high computing requirements, they fall into the zero-delay segment. Because they are portable devices, they fall into the zero-power segment. We all want cheap, highly capable devices that give us instant answers and that work on weak ambient light. The overlap of the zero-cost, zero-delay, and zero-power segments is the *leading-edge wedge*. Figure 6 illustrates the leading-edge wedge.

As Figures 5 and 6 show, the leading-edge wedge is a tiny percentage of the embedded systems market, but it is growing rapidly, and it can have better margins than most embedded applications. The leading-edge wedge is the future for embedded systems applications. Tomorrow's engineers will have to design applications to meet the requirements of this wedge. Engineers will struggle with leading-edge wedge designs unless universities provide their students with the necessary tools, which they don't today. Today's universities teach two methods for designing applications: logic design (fixed resources and fixed algorithms) and microprocessor-based design (fixed resources and dynamic algorithms). This won't do for applications in the leading-edge wedge. A look at programmable logic devices shows why.

Dynamic Logic

Invention of the programmable logic device (PLD) by Sven Whalstrom was a conceptual breakthrough (as was the computer). Fortunately for PLD companies, Sven, whose fundamental patent (#3,473,160, Electronically Controlled Microelectronic Cellular Logic Array) was filed in 1966, was too early. His conceptual breakthrough came before the semiconductor fabrication process made the concept practical. At about the time Sven's patent expired, transistors available on a single die rose to the point of practical application and companies such as Altera (1983) and Xilinx (1984) began to develop the market for Sven's breakthrough.

The PLD is conceptually a two-layer device. The first layer consists of logic elements and an interconnect structure. The second layer contains memory. Values in the memory cells establish connections between the logic elements and the interconnect structure. These values personalize the device. Figure 7 shows the structure of a programmable logic device. The programmable logic device allows the engineer to select both the resources and the algorithm. Both the resources and the algorithms may vary over time to solve the problem. This is dynamic logic: *dynamic resources and dynamic algorithms.*

Dynamic resources and dynamic algorithms can be more efficient than fixed resources and dynamic algorithms,

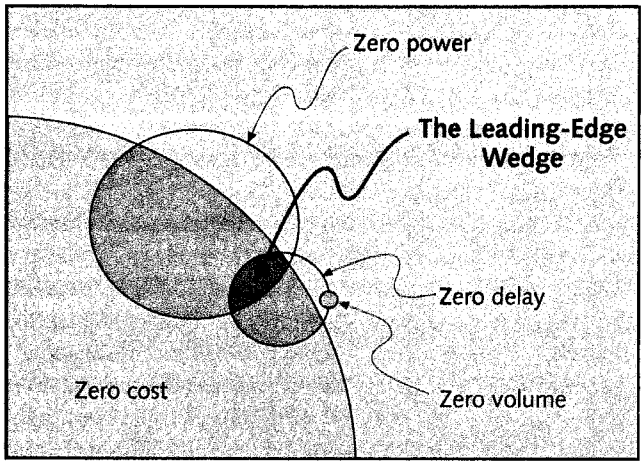

Figure 6. The leading-edge wedge is the overlap of the zero-cost, zero-delay, and zero-power segments of the embedded systems market.

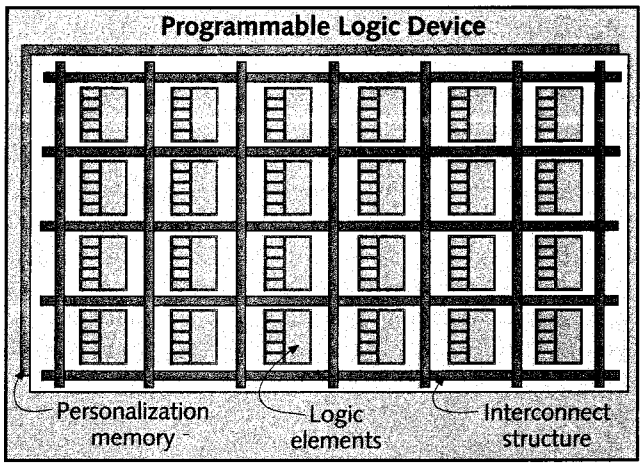

Figure 7. The programmable logic device comprises a layer of logic and interconnect plus a layer of memory.

References

- Church, Charles C. "Computer instruction repertoire—Time for a change." *AFIPS Conference Proceedings*, V. 36, Montvale, NJ, 1970, p. 343–349.
- Claire, C. *Designing Logic Systems Using State Machines*. McGraw-Hill, New York, NY, 1973.
- Ditzel, David R. "Why RISC Has Won." *The SPARC Technical Papers*, edited by Ben J. Catanzaro. Springer-Verlag, New York, NY, 1991, p. 67–70.
- Glushkov, V. "Automata theory and formal microprogram transformation." *Kibernetika*, V. 1, 1965, p. 1–9.
- Gwennap, Linley, and Kevin Krewell. *Intel Microprocessor Forecast 1H00*. MicroDesign Resources, Sunnyvale, CA, 2000.
- *The 8086 Family User's Manual*. Intel Corp., Santa Clara, CA, 1979.
- Karp, R., and R. Miller. "Parallel programming schemata." *S. Computer System Science*, V. 3, May 1969, p. 147–195.
- Sipper, Moshe, and Eduardo Sanchez. "Configurable Chips Meld Software and Hardware." *Computer*, V. 33, No. 1, January 2000, IEEE, Piscataway, NJ, p. 120–121.

Recommended Reading

- Mackay, Charles. *Extraordinary Popular Delusions and the Madness of Crowds*. Richard Bentley, Publisher in Ordinary to Her Majesty, London, 1841.
- Rand, Ayn. *The Fountainhead*. International Collectors Library, Garden City, NY, 1943 and 1968.

because the engineer can choose the appropriate resources for the problem and can build a state sequencer exactly matched to the algorithm. In a microprocessor, the algorithm is buried in a program that is compiled into an instruction set. The instruction set drives the microprocessor's state sequencer. The state sequencer manipulates the microprocessor's hardware resources to run the programmed solution. Each translation (algorithm, program, compiler, instruction set, state sequencer) lowers the efficiency. Efficiency is also sacrificed if the microprocessor's resources don't exactly match the problem's requirements.

With a dynamic logic solution, the engineer can choose different resources and different algorithms for subsets of the problem. The resources and the algorithms can vary over time to solve the problem. In the past, the difficulty with PLDs was that they were slow to configure, were limited in capacity, and had more overhead than the U.S. government (20 transistors in the device yielded 1 user-level transistor). Semiconductor fabrication continues to improve, however, and today's PLDs are faster to configure and have sufficient capacity to overcome their overhead, even for leading-edge applications. This overhead seems to be a big problem. It isn't. Power dissipation and performance are important for leading-edge-wedge applications. Absolute transistor-count efficiency is not the primary concern. Also, today's commercial PLDs are designed for general-purpose applications and for prototyping. Programmable logic designed into leading-edge-wedge applications can improve transistor-count efficiency by implementing flexibility appropriate to the application. For example, a programmable logic device designed to displace the DSPs and ASICs in 100 million cellular phones won't have 95% overhead transistors. Startups such as QuickSilver, Triscend, Chameleon Systems, and Morphix are betting that this is so as they enter the market for dynamic-logic applications.

Isn't This Like...

The computer industry, in the days of batch processing, experimented with "dynamic microprogramming"—a concept akin to dynamic logic. Dynamic-microprogramming theory held that it would be more efficient to change the instruction set of the computer (by downloading custom microcode) to match the language of the target programs than to run a single instruction set for all languages. For programs in Fortran, the computer loaded microcode for a Fortran-oriented instruction set. For Cobol, the computer loaded microcode for Cobol. For Algol, the computer loaded microcode for Algol. The computer's instruction set changed to meet the needs of the programming language. The overhead to load the language-oriented microcode had to be less than the inefficiency of running a single instruction set on all languages. The concept may have worked in a batch-processing environment as memories got faster, but multitasking killed dynamic microprogramming; there was no way to load custom microcode efficiently for time slices.

The Battle Between Microprocessors and PLDs

Logic-macro functions have lost the battle for the majority of applications in the problem domain. Microprocessors dominate applications in embedded systems and therefore, in leading-edge wedge applications. In the near future, microprocessors will fight to hold onto leading-edge-wedge applications—against encroachment by PLD-based dynamic-logic implementations.

Microprocessors are in the wrong place, and they are headed in the wrong direction. Microprocessors are in the wrong place now, because, as integrated circuits, they are designed to run an entire instruction set rather than simply implement in hardware one algorithm for a single application. They give up efficiency because they try to meet the requirements of a range of applications by putting a general-purpose instruction set between the hardware and the problem. Microprocessors are headed in the wrong direction, because they are going down the path of using arcane, convoluted schemes to process general-purpose

instructions faster. As they get faster and more complex, microprocessors use more power (though progress in semiconductor fabrication offsets this). The market for microprocessors will continue to grow, however, because they gain applications at the high end (zero-delay segment) and at the low end (zero-cost segment), even as they lose applications in the leading-edge wedge.

A custom-designed programmable logic device can be dynamically configured to provide just the right resources at the right time for its intended application. It can be designed so that the resources and the algorithm dynamically adjust to the needs of the application. One issue that will decide whether it is microprocessors or PLDs that win applications in the leading-edge wedge is computational power efficiency. Computational power efficiency is the product of the computing done and the power used. Better computational power efficiency gets more done for less power.

The cellular telephone, for example, does a variety of compute-intensive tasks in the course of a call (call setup and tear down, encoding, decoding, etc.). A microprocessor-based cellular telephone trades efficiency in each of these application segments for the ability to deal with all of them. A PLD could be configured with the proper resources to meet each task efficiently. Both the PLD and the microprocessor have the capability to do the application. One issue is whether reconfiguring the PLD gives up more power than the microprocessor gives up in lack of efficiency. For most applications, dynamically reconfiguring the PLD doesn't use significant power. A custom programmable logic device would be substantially more efficient than the microprocessor for some high-volume leading-edge-wedge applications.

However, the computational power efficiency of the PLD is only one issue involved in whether it overtakes the microprocessor for leading-edge-wedge applications. A second major issue is whether there will be engineers with the skills to solve problems with dynamic logic.

Universities teach logic-design methods (fixed resources and fixed algorithms), and they teach computer-based design methods (fixed resources and dynamic algorithms), but they do not teach dynamic logic (dynamic resources and dynamic algorithms). Courses at many universities use PLDs. In these courses, the PLDs are a laboratory aid in teaching logic design methods (fixed resources and fixed algorithms) or in teaching computer architecture (fixed resources and dynamic algorithms). Students build logic designs or implement instruction sets using PLDs, but it is unlikely that students will be taught dynamic logic methods. There currently are no textbooks, no application notes, and no instructors—there's no demand from industry (except for a few startups) for dynamic logic design skills. It is ironic that the building block for dynamic logic applications (the PLD) is an important tool for teaching competing design methods. Students use PLDs to build prototype direct hardware implementations and to build prototype CPU designs. In fairness to the universities, this is also how most practicing engineers use PLDs today. The PLD's use as a teaching tool and for prototyping has created educational and cultural barriers to its direct use in dynamic logic applications.

Problem Solving Method	Resources	Algorithm
Direct Hardware	Fixed	Fixed
Microprocessor	Fixed	Dynamic
Dynamic Logic	Dynamic	Dynamic

Table 1. Next-generation problems will best be solved using dynamic resources and dynamic algorithms.

Summary

Table 1 shows a summary of problem-solving methods.

In a 40-year infatuation with the computer, universities have graduated generations of engineers with the skills to solve computer-based problems. ("Give me a microprocessor and I'll program it to solve any problem.") The rapid growth of applications for microprocessors has reinforced computer-based problem-solving methods (fixed resources and dynamic algorithms) for generations of engineers, and it has left us devoid of skills to meet the future.

A recent article by Sipper and Sanchez in *Computer* ("Configurable Chips Meld Software and Hardware") points to the need for a change: "Developments in configurable computing increasingly blur the line between hardware and software, a trend that represents a major shift in computing practice. To keep their offerings current and relevant, universities should modify their computer science curricula to better prepare students for this new era."

This suggestion may take us in the wrong direction. The orientation to "configurable computing" ties dynamic logic to computing. That may be the only practical way to transition from microprocessor-based design methods to methods that include some dynamic logic, but it burdens the solutions with a restrictive computing-oriented legacy.

We believe dynamic logic should be taught as an extension of logic design. To learn logic design, a student must practice selecting the hardware resources appropriate to the problem and must map the algorithm and resources together with a derived state sequencer in a direct hardware implementation. Dynamic-logic methods extend logic design to include the element of time. Direct-hardware solutions are invariant over time. Dynamic-logic solutions vary with time. A dynamic-logic solution is like a "paged" direct hardware solution. Each segment of the problem could be treated as a direct hardware solution, with its own hardware resources and its own state sequencer, both designed to meet the unique requirements of a particular segment of the problem. The solution would be paged into the hardware at the appropriate time by reconfiguring the underlying programmable logic devices. Logic design courses already teach two components of the design method: selecting the appropriate resources and mapping algorithms into hardware. Computer courses teach the third component: dynamic use of resources. ◇