Flexible Architectures for Engineering Successful SOCs

Chris Rowen
Tensilica, Inc.
3255-6 Scott Blvd.
San Jose, CA 95054
(408) 986-8000
rowen@tensilica.com

Steve Leibson
Tensilica, Inc.
3255-6 Scott Blvd.
San Jose, CA 95054
(408) 986-8000
sleibson@tensilica.com

ABSTRACT

This paper focuses on a particular SOC design technology and methodology, here called the advanced or processor-centric SOC design method, which reduces the risk of SOC design and increases ROI by using configurable processors to implement on-chip functions while increasing the SOC's flexibility through software programmability. The essential enabler for this design methodology is automatic processor generation—the rapid and easy creation of new microprocessor architectures, complete with efficient hardware designs and comprehensive software tools. The high speed of the generation process and the great flexibility of the generated architectures underpin a fundamental shift of the role of processors in system architecture.

Categories and Subject Descriptors

B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids

General Terms

Design, Economics, Languages, Verification.

Keywords

RTL, SOC, MPSOC, processor cores, RISC.

1. INTRODUCTION

The use of tailored processors is the foundation of an advanced SOC design method—first, the mapping of a system's functions into a set of communicating tasks; second, the combined optimization of each task and the processor architecture on which it runs; and third, the integration of the processors and task software into a fast,accurate system model and corresponding integrated-circuit implementation.

2. Processor-Centric SOC Design

Deep-sub-micron silicon opens up the possibility of highly concurrent systems architectures—architectures that improve overall throughput, latency, and efficiency by executing many tasks and operations in parallel. The target applications for volume SOC designs—communications, consumer, and computation systems—often show high degrees of intrinsic concurrency, including parallelism at the task level (major subsystems that can run in parallel with others) and at the data and instruction level (individual operations within one task that can run in parallel with one another). The processor-centric SOC design methodology seeks to exploit parallelism at the task level. Use of optimized, automatically generated processors seeks to exploit the parallelism inherent at the data and instruction level.

Before launching into a detailed discussion of the mechanisms of the new SOC methodology, a short discussion of the impact of processor configuration on application performance will show the core benefits of processor optimization. The hardware-centric and software-centric views of SOC design require parallel discussion, because these two traditions often have wildly different perspectives on development flow, measures of goodness, and mechanisms for change. In fact, hardware and software engineers often see distinctly different motivations for migrating specific functions into an application-specific processor block.

SOC designers face a major dilemma as they architect and implement a system. Many sub-systems can be implemented, at least in theory, as either a block of hardwired logic or as software running on a general-purpose processor. For many data-intensive functions, the hardwired logic implementation can be more than one hundred times smaller or faster than the software implementation because the logic contains only the data path elements required and these elements can often be easily arranged to run in parallel for high throughput.

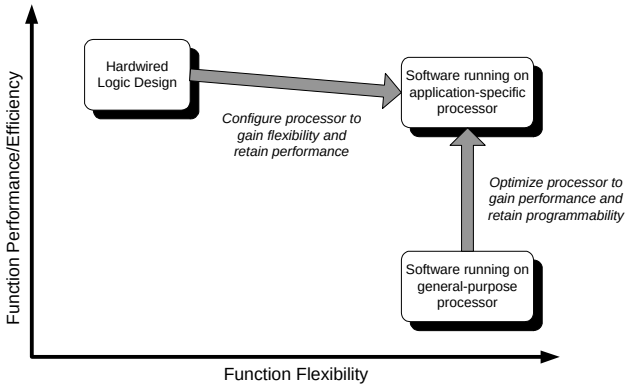

Figure 1. Migration from hardwired logic and general-purpose processors to application-specific processors

DAC 2004, June 7–11, 2004, San Diego, California, USA

By contrast, processors show their superior flexibility when function changes are required. Modifying and retesting a software change may be more than one hundred times faster and cheaper than modifying the circuits on a large IC. This contrast is shown in Figure 1. The implementation of the function on an application-specific processor retains all of the programming flexibility of a general-purpose processor but with functional performance or efficiency that rivals that of hardwired logic. Figure 1 also suggests the benefits of using application-specific processors seen by hardware designers and by software developers.

Hardware designers move from hardwired logic to application specific processors to reduce development time and ensure continuous and easy upgradeability as the application evolves. Customizing and extending predefined base processor architectures instead of designing complex logic blocks reduces the initial SOC development effort, because specification of processor configurations and software is easier than specification of logic design at the gate or RTL level. Processor-based design of logic functions reduces risk in two ways. The initial design is easier to verify, because the specification is more concise and can be simulated in the system context more directly and at far higher speeds. Secondly, the resulting design is more tolerant of specification changes because far more of the design, especially the control flow of the design, is implemented in software and can be changed at any time, before or after fabricating the SOC. Simplified design verification is a major incentive for adoption of configurable processors over traditional RTL design.

Software developers move to application-specific processors from general-purpose processors to reach performance and efficiency goals without giving up the well-known benefits of rapid software evolution provided by a programmable processor. Software-based SOC design gets the software team involved earlier, and gives the team a greater role in the overall project. In many cases, the tools for processor generation give the software developer sufficient insight into both application performance and hardware implementation that they can play a central role even in jobs formerly reserved for specialized hardware designers. This migration from hardwired logic and from software running on general-purpose processors is governed by different situations and offer different, but important, benefits to their users.

3. Accelerating Processors for SOC Design

As the complexity of embedded systems rises, software naturally takes on a more important role in the system. As the product design evolves and requires more performance or more functions running in software, every software component must run faster. Not every line of code, of course, is equally important. Some tasks, and some task sections, dominate the software performance profile. If those sections and those tasks can be effectively optimized, the overall system performance improves.

3.1 The Evolution of Generic Processors

Most microprocessor architectures were created with general-purpose applications in mind. These architectures are often good targets for a broad range of generic integer applications written in C or C++, but they are typically capable of only modest data-manipulation rates. Such one-size-fits-all architectures may be appropriate for high-level control tasks in SOC applications, but they are significantly less efficient for more data-intensive tasks such as signal, media, network, and security applications.

The origins and evolution of microprocessors further constrain their use in traditional SOC design. Most popular embedded microprocessors, especially the 32-bit architectures, descend directly from 1980s desktop computer architectures such as ARM (originally the "Acorn RISC Machine, a British desktop), MIPS, 68000/ColdFire, PowerPC, and x86. Designed to serve general-purpose applications, these processors support only the most generic data types such as 8-, 16-, and 32-bit integers. Likewise, they support only the most common operations such as integer load, store, add, shift, compare, and bit-wise logical functions.

Their general-purpose nature makes these processors well suited to the diverse mix of applications run on computer systems: their architectures perform equally well when running databases, spreadsheets, PC games, and desktop publishing. However, all these general-purpose processors suffer from a common bottleneck. Their need for complete generality requires them to execute an arbitrary sequence of primitive instructions on an unknown range of data types. Put another way, general-purpose processors are not optimized to deal with the specific data types of any given embedded task. Inefficiencies result.

Of course, general-purpose processors can emulate complex operations on application-specific data-types using sequences – sometimes long sequences - of primitive integer operations. For example, Section 5.1 shows how the basic pixel `blend` operation, arguably a single operation for an imaging-oriented architecture, requires 33 RISC operations to execute. Many embedded applications can be expressed and implemented most naturally in something other than 32-bit integer operations. Security processing, signal processing, video processing, and network protocols all have unique computational requirements with, at best, a loose fit to basic integer operations. This "semantic gap" has long inspired efforts in application-directed processor architecture, but it has rarely been economical to commercialize such processors because of the high costs and specialized skills involved.

3.2 Specialized Processors for Special Needs

Compared to general-purpose computer systems, embedded systems comprise a more diverse group and individually show more specialization. A digital camera must perform a variety of complex image-processing tasks but it never executes SQL database queries. A network switch must handle complex communications protocols at optical interconnect speeds but it doesn't manipulate 3D graphics images.

The specialized nature of each individual embedded application creates two issues for general-purpose processors in data-intensive embedded applications. First, the critical data-manipulation functions of many embedded applications and a processor's basic integer instruction set and register file are a poor match. Because of this mismatch, these critical embedded functions require many computation cycles when run on general-purpose processors.

Second, more focused embedded products cannot take full advantage of a general-purpose processor's broad capabilities. Expensive silicon resources built into the processor go to waste because the specific embedded task that's assigned to the processor doesn't need them. Unused features that might be tolerable within the cost and power budgets of a desktop computer are a painful extravagance in low-cost, battery-powered consumer products.

Many embedded systems interact closely with the real world or communicate complex data at high rates. A hypothetical general-purpose microprocessor running at tremendous speed could perform these data-intensive tasks. This is the basic assumption behind the use of multi-GHz processors in today's PCs: throw a fast enough processor at a problem (no matter the cost in dollars or power dissipation) and you can solve any problem. For many embedded tasks, however, no such processor exists as a practical alternative because the fastest available processors typically cost orders of magnitude too much and dissipate orders of magnitude too much power to meet embedded-system design goals. Instead, embedded-system hardware designers have traditionally turned to hardwired circuits to perform these data-intensive functions.

3.2.1 Configurability and Extensibility

Changing the processor's instruction set, memories and interfaces can make a significant difference in its efficiency and performance, particularly for the data-intensive applications that represent the "heavy lifting" of many embedded systems. These features might be too narrowly used to justify inclusion in a general-purpose instruction set, hand-designed processor hardware, and hand-crafted software tools.

The general-purpose processor represents a compromise where features that provide modest benefits to all customers supercede features that provide dramatic benefits to a few. This design compromise is necessary because the historic costs and difficulty of manual processor design limit the number of different processor designs that can be completed. Automatic processor generation reduces the cost and development time so that inclusion of application-specific features and deletion of unused features suddenly becomes attractive.

We use the term configurable processor to denote a processor whose features can be pruned or augmented by parametric selection. Configurable processors may be implemented in many different hardware forms, ranging from ASICs with hardware implementation times of many weeks, to FPGAs with implementation times measured in minutes. An important superset of configurable processors is extensible processors—processors whose functions, especially the instruction set, can be extended by the application developer to include features never considered by the original processor designer.

For both configurable and extensible processors, the usefulness of the configurability and extensibility is strongly tied to the automatic availability of both hardware implementation and software environment supporting all aspects of the configurations or extensions. Automated software support for extended features is especially important, however. Configuration or extension of the hardware without complementary enhancement of the compiler, assembler, simulator, debugger, real-time operating systems, and other software support tools would leave the promises of performance and flexibility unfulfilled because the new processor could not be programmed.

3.2.2 Processor Extensibility

Extensibility's goal is to allow features to be added or adapted in any form that optimizes the cost, power, and application-performance of the processor. In practice, the configurable and extensible features can be broken into four categories, as shown in Table 1 below.

Table 1. Processor configuration and extension types

Instruction Set	Memory System	Interface	Processor Peripherals
• Extensions to ALU functions using general registers (e.g. population count instruction) • Coprocessors supporting application-specific data types (e.g. network packets, pixel blocks), including new registers and register files • Wide instruction formats with multiple independent operation slots per instruction • High-performance arithmetic and DSP (e.g. compound DSP instructions, vector/SIMD, floating point), often with wide execution units and registers • Selection among function unit implementations (e.g. small iterative multiplier vs. pipelined array multiplier)	• Instruction-cache size, associativity, and line size • Data-cache size, associativity, line size, and write policy • Memory protection and translation (by segment, by page) • Instruction and data RAM/ROM size and address range • Mapping of special-purpose memories (queues, multi-ported memories) into the address space of the processor • Slave access to local memories by external processors and DMA engines	• External bus interface width, protocol, and address decoding • Direct connection of system control registers to internal registers and data ports • Arbitrary-width wire interfaces mapped into instructions • Queue interfaces among processors or between processors and external logic functions • State-visibility trace ports and JTAG-based debug ports	• Timers • Interrupt controller: number, priority, type, fast switching registers • Exception vectors addresses • Remote debug and breakpoint controls

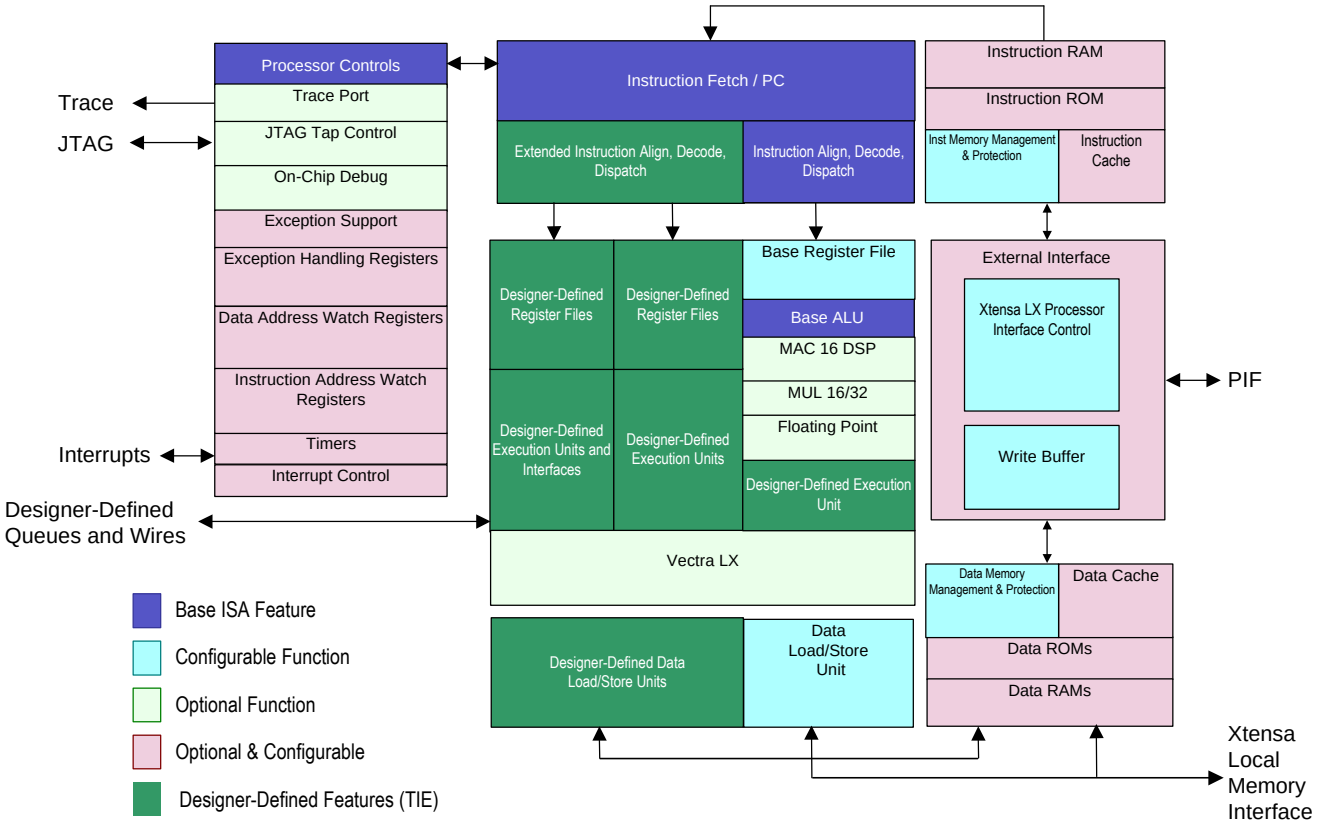

Figure 2. Block diagram of Configurable Xtensa Processor

A block diagram for a configurable processor is shown in Figure 2, again using Tensilica's Xtensa processor as an example. The figure identifies baseline instruction-set architecture features, scaleable register files, memories and interfaces, optional and configurable processor peripherals, selectable DSP coprocessors, and facilities to integrate user-defined instruction-set extensions. Almost every feature of the processor is configurable, extensible, or optional (including the size of almost every data path, the number of type of execution units, the number and type of peripherals, the number and size of load/store units and ports into memory, the size of instructions, and the number of operations encoded in each instruction).

3.3 Designer-Defined Instruction Sets

Processor extensibility serves as a particularly potent form of configurability because it handles a wide range of applications and is easily usable by designers with a wide range of skills. Processor extensibility allows a system designer or application expert to directly exploit proprietary *insight* about the application's functional and performance needs directly in the instruction-set and register extensions. This shift of insight, however, also makes special demands on the processor-generator environment. The means of expressing extensions must be flexible—to accommodate a wide range of possible instruction-set extensions. The instruction-set description tools must be intuitive and bullet-proof—to ensure that someone who is not a processor designer can easily create a new instruction set with full software support. The instruction set description method must prevent the creation of instructions with subtle bugs that could prevent the basic processor from functioning correctly.

A simple example written in the Tensilica Instruction Extension (TIE) language illustrates the potential simplicity of this approach. Figure 3 shows the complete TIE description of an instruction that takes two sets of four 8-bit values packed into two 32-bit entries (`a` and `b`) in the primary (`AR`) register file, multiplies together the corresponding 8-bit values, and sums adjacent 16-bit multiplication results into a pair of 16-bit values. The resulting value (`c`) is written back into a third 32-bit `AR` register file entry:

```
operation mac4x8 {in AR a, in AR b, out AR c} {} {
assign c = {a[31:24]*b[31:24] + a[23:16]*b[23:16], a[15:8]*b[15:8] + a[7:0]*b[7:0]};}
```

Figure 3. Simple 4-way MAC TIE example

From this instruction description, the following actions take place automatically:

- new data path elements, including four 8x8 multipliers and two 16-bit adders are added to the processor hardware description
- new decode logic is added to the processor to decode the new `mac4x8` instruction using a previously unallocated operation encoding
- the integrated development environment, including instruction-set simulator, debugger, profiler, assembler, and compiler are extended to support this new `mac4x8` instruction
- plug-in extensions for third-party tools, including debuggers and RTOS are generated so these tools also include support for the new instruction

Instruction-set-description languages such as TIE represent an important step beyond traditional hardware description languages (HDL) such as Verilog and VHDL. Like HDLs, the semantics of instruction-set-description languages are expressed as a combination of logical and arithmetic operations on bits and bit-fields transforming input values into output values.

The TIE language, for example, uses pure combinational Verilog syntax for describing instructions. Unlike HDLs, key software information about instruction names, operands, and pipelining must be succinctly expressed. This form of expression raises the level of abstraction for design by direct incorporation of new hardware functions into the processor structure and directly exposes the new instruction(s) in the high-level programming environment available to the software team.

3.4 System Design with Multiple Processors

Semiconductor device scaling creates tremendous opportunity for high density and high parallelism for SOC devices. The effectiveness of extensible processors both in traditional processor roles and as replacements for hardwired logic enables a design style in which large numbers of processors operate in parallel to efficiently implement the rich functions of the system. The mere availability of such small fast processors does not solve the system-design problem however. We must consider how multiple processors can work together on complex problems.

3.4.1 Available concurrency

The steady proliferation of digital electronics in computing, communication, and consumer applications is strongly tied to the steady progress of semiconductor device scaling. Electronic product performance benefits from improving the speed of individual transistors. The bigger benefit, however, derives from integration of large numbers of transistors together in each integrated circuit. Progress in digital electronics, then, depends on the ability of the chip or system designer to find ways to use many transistors, working concurrently, to more efficiently implement the system's functions. The designer can exploit parallelism at many different levels, but all of these levels can be reduced or generalized to three:

- **Bit-level parallelism:** Almost all digital systems operate on data (text characters, pixels, voice samples, network packet headers) with more than one bit of precision. Many transistors—many gates—operate in parallel to perform the basic operations on this data—add, shift, bit-wise "and", load, store, compare, and so forth. The density of today's silicon technology is already so high that we rarely need to think about the bit-level parallelism, except when the natural computation is unusually wide. Both existing processors and logic design methods are well-equipped to deal with bit widths of 32 or 64 bits. However, some encryption and networking cases arise where the natural bit width of the task to be performed is much wider.
- **Operation-level parallelism:** Any complex function consists of a series of operations performed on groups of data values. Some of those operations are intrinsically dependent on one another and must be performed serially. In almost any function, however, some dependencies are somewhat looser and it is theoretically possible to perform loosely dependent operations in parallel. The system designer may still choose to perform some non-dependent operations serially, either to share the hardware, or to simplify the computation. Microprocessors have traditionally served as a simple way to serially execute almost any digital function. Processor configuration and extension principally serve as a simple, structured way to increase operation-level parallelism. The primary techniques for increasing operation-level parallelism include deep operation-unit pipelining, multiple instruction issue (VLIW or superscalar) and single instruction, multiple data (SIMD) architecture.
- **Task-level concurrency:** Most systems perform more than one essential function. These functions or subsystems may communicate intermittently with others but they are sufficiently independent that their functions can be described and implemented separately from other functions in the system. The collection of communicating functions in a system—user interface, video processing, audio processing, wireless channel processing, and encryption functions in a cell-phone for example—are sufficiently independent that the system user wants the effect or illusion of simultaneous parallel operation by all. Semiconductor device scaling enables the combining of all these functions into a single device, so task-level parallelism is increasingly important to the chip architect. Task-level parallelism is most easily exploited when it is already explicit in the functions of the system—e.g. the audio and video subsystems obviously run in parallel. Task-level parallelism can also be extracted from applications originally developed as a single sequential task. No universal extraction method exists, but appropriate tools and building blocks help in discovery of hidden task-level parallelism.

3.4.2 Parallelism and power

Parallelism also provides one key to power efficiency in SOC devices. Consider a CMOS circuit in which most of the power dissipation is active power. In this case, the power dissipation for the circuit is roughly

$$P \propto CV^2 f$$

where C is the switched capacitance of the circuit, V is the supply voltage and f is the effective switching frequency of the nodes in the circuit. We can take advantage of the fact that the maximum operating frequency of the circuit is roughly proportional to voltage (a good assumption for typical CMOS circuits). If the function implemented in the circuit can be scaled by some factor α (>1), so that the circuit has more transistors (for more parallelism) by a factor of α, but can achieve the same throughput at frequency reduced by α, we can reduce the operating voltage by roughly α as well, so we see a significant reduction in power:

$$P_{new} \approx (aC)\left(\frac{V}{a}\right)^2\left(\frac{f}{a}\right) = \frac{P_{original}}{a^2}$$

In practice, α cannot be increased arbitrarily because of the operation of CMOS circuits degrades as operating voltage approaches transistor threshold voltage. Put another way, there's a practical upper limit to the size of α. On the other hand, the scaling of switched capacitance (aC) with performance may be unrealistically conservative. As shown in the EEMBC benchmarks cited earlier, small additions to the processor hardware, which slightly increases switched capacitance, often leads to large improvements in throughput. This simplified analysis underscores the important trend towards increased parallelism in circuits.

3.5 A Pragmatic View of Multiple Processor Design Methodology

In the best of all possible worlds, applications developers would simply write algorithms in a high-level language. Software tools would identify huge degrees of latent parallelism and generate code running across hundreds or thousands of small processors. Hardware design would be trivial too, because a single universal processor design would be replicated as much as necessary to balance cost and performance goals.

Unfortunately, this view is pure fantasy for most applications. Latent parallelism varies widely across different embedded systems and even when parallelism exists, no fully automated methods are available to extract it. Moreover, a significant portion of the available parallelism in SOC applications comes not from a single algorithm, but from the collection of largely-independent algorithms running together on one platform. Developers start from a set of tasks for the system and exploit the parallelism by applying a spectrum of techniques, including four basic actions:

1. Allocate (mostly) independent tasks to different processors, with communications among tasks expressed via shared memory and messages.
2. Speed up each individual task by optimizing the processor on which it runs. Typically this process involves processor extension to create an instruction set and a program that performs more operations per cycle (more fine-grained parallelism).
3. For particularly performance-critical tasks, decompose the task into a set of parallel tasks running on a set of communicating processors. The new suite of processors may all be identical and operate on different data subsets or they may be configured differently, each optimized for a different phase of the original algorithm.
4. Combine multiple tasks together on one processor by time-slicing. This approach degrades parallelism, but may improve SOC cost and efficiency, if a processor has available computation cycles.

These methods interact with one another, so iterative refinement is probably essential, particularly as the design evolves. As a result, quick exploration of tradeoffs through trial system design, experimental processor configuration, and fast system simulation are especially important.

3.6 Build Optimized Platforms to Aggregate Volume

Inadequate volume is the bugbear of SOC economics. SOC integration can easily drive down electronics manufacturing costs compared to board-level integration, but design and prototyping costs can be significant, especially for designs with less than a million units of volume. The intrinsic flexibility of designs using programmable processors as their basic fabric increases the number of systems that can be supported by one chip design.

Figure 4 shows a simple model of the total chip cost, including amortized development costs, and the impact of increasing the number of systems design supported by one SOC design. The model assumes $10,000,000 total development cost, $15 chip manufacturing cost, and shows results for 100,000-unit and 1,000,000-unit production volumes. This model also assumes an incremental chip cost (5%) for the small overhead of programmability required to allow the SOC to support more than one system design.

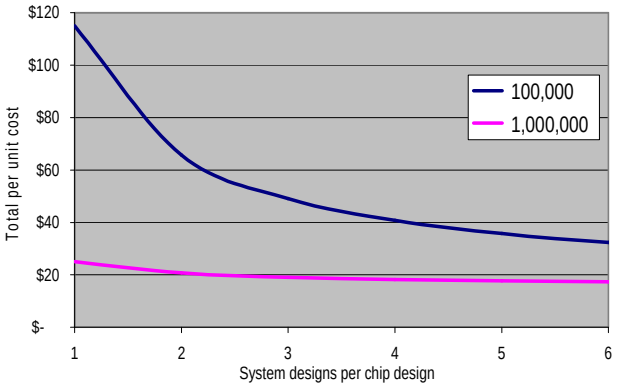

Figure 4. Example Amortized Chip Costs (100K and 1M system volumes)

As the figure shows, programmability is useful for all SOCs and is economically essential for lower volume SOCs.

4. ACKNOWLEDGMENTS

This paper is based on Chris Rowen's book, *Engineering the Complex SOC*, to be published this month by Prentice Hall.