CHAPTER 3

A New Look at SOC Design

This book focuses on a particular SOC design technology and methodology, here called the *advanced* or *processor-centric SOC design method.* The essential enabler for this technology 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. Automatic generation allows processors to be more closely tailored to the computation and communication demands of an application. This tailoring, in turn, expands the roles of processors in system architecture. The processors may serve both as a faster or more efficient alternatives to processors used historically in programmable roles and as a programmable alternative to other forms of digital logic, such as complex state machines. The continued rapid pace of transistor scaling allows a rich and diverse set of functions to be implemented together on SOCs. When these SOCs are designed with configurable processors, the resulting electronic systems have better performance, better efficiency, and better adaptability.

This chapter introduces the basics of multiple-processor system design and automatic processor generation, shows the impact of processor tailoring on tasks historically implemented as software on general-purpose processors, and sketches the influence of tailored processors on tasks traditionally implemented as nonprogrammable hardware blocks. The use of tailored processors is the foundation of the advanced SOC 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 properly operating system model and corresponding integrated-circuit implementation.

3.1 The Basics of Processor-Centric SOC Architecture

Deep-submicron silicon opens up the possibility of highly parallel 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 parallelism, 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.

3.1.1 Processor Generation

Chapters 5, 6, and 7 focus on the single most important component of the new SOC design flow—the design of the individual processors that comprise the heart of the SOC design. A processor design has three important dimensions:

1. **Architecture:** Defines resources available to the programmer, including the instruction set, registers, memories, operators, interrupt- and exception-handling methods, and system-control functions. The architecture can be considered a contract between the hardware design and the software environment. In SOC design, the "goodness" of a processor architecture is not measured by abstract notions of performance and generality. Instead, architecture quality can be quantified by measurements of efficiency and performance of applications that are expected to run on the processor.
2. **Hardware implementation:** The logic and circuit design that implements the fundamental process of fetching and executing instructions. The hardware implementation includes the microprocessor pipeline; data paths and widths; and the arrangement of memories, buses, and other interfaces to circuits outside the processor. The mapping of this logic into transistor circuits and integrated circuit structures plays a central role in determining the cost, performance, and power dissipation of the final hardware.
3. **Software environment:** The collection of software tools used to develop high-level-language programs and to translate those programs into a correct sequence of instructions, plus runtime software components to support application execution on the architecture. Tailoring the processor (or set of processors) to the intended application is central to the SOC development methodology, so the software environment must include cycle-accurate instruction set simulators, code performance profilers and interprocessor communications modeling tools.

Sometimes it is also useful to speak of a *microarchitecture*, meaning key hardware implementation choices such as the arrangement of the execution pipeline, the size and width of cache memories, and other major determinants of processor performance.

Architecture, but not microarchitecture, is directly visible to the programmer. Microarchitecture, however, may have an important indirect impact in that it will affect application performance and hardware cost. Both architectural and microarchitectural changes may be important levers for optimizing the processor to meet cost and performance goals.

The essence of automatic processor generation is the automated translation of a high-level, abstract processor description into a complete hardware implementation and tailored software environment. The processor description may start from a base architecture—a simple standard set of architectural features useful across all applications—and add feature enhancements needed by the target set of applications. Starting from a simple base-processor architecture offers two important benefits:

1. Programs can be written for the base architecture and run unmodified on any configuration of the processor. This characteristic is potentially important for basic software utilities that are widely needed but not performance-critical.
2. Having a small but functional minimum processor configuration simplifies the generation of compilers, simulators, debuggers, and operating systems because the most basic operations—load, store, add, branch—are known to exist in all configurations.

The processor description constitutes sufficient information for the automatic generation of a quality hardware implementation and tailored compilers, assemblers, debuggers, and simulators and to adapt real-time operating systems to the new architecture. The automatic generation of the hardware design and software tools also enables rapid assessment of the specified architecture's suitability for the target set of applications by enabling quick prototyping and measurement of real implementation characteristics.

Rapid measurement of hardware size, power, and speed—and rapid performance profiling of the application set on the target architecture—yield the essential insight into design suitability with regard to cost, power, and speed. These traits of the advanced SOC design method allow the designer both to rapidly iterate through many potential processor architectures and to start the integration of this processor subsystem into the SOC. The basic structure of the processor optimization flow is shown in Fig 3-1.

The two major inputs into the processor-generation flow are the application source code and sample input data for that application. The generator's output is the processor design and configured software tools tuned for that processor configuration. The basic process consists of generating an initial processor with its software environment, then compiling the application source code using that software environment to create a machine-code binary program, and then simulating the application with that binary program and sample application input data. This process yields a performance profile and corresponding output data.

The performance profile indicates the number of cycles required to execute the application: one of the key measures of processor effectiveness. This design process also generates the processor's RTL implementation and scripts for gate-level design tools, so the netlist and physical placement and routing of gate-level cells can be produced. Clock frequency, silicon chip

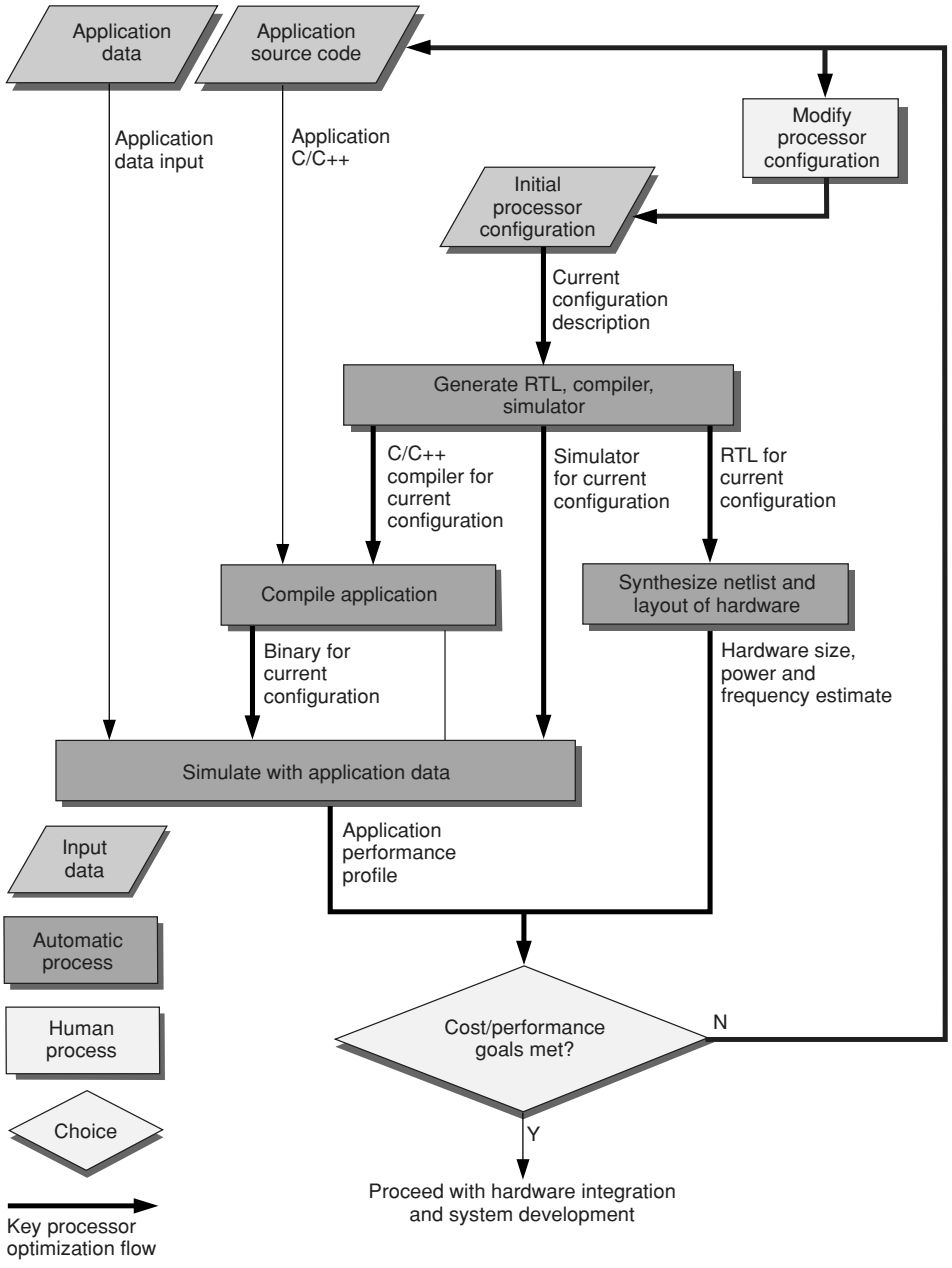

Figure 3-1 Basic processor generation flow.

area, and power dissipation can be accurately estimated from the physical design. These hardware implementation characteristics represent the other key measures of the processor configuration's appropriateness.

If the cost or power dissipation is too high, the configuration can be downsized. If the number of execution cycles is too high, adding new instructions, increasing memory capacity or bus width, or making other changes to the processor's definition can enhance the configuration's efficiency to meet performance goals. This enhanced processor configuration serves as the seed for generating updated compilers, the RTL description, and all their by-products so that the design team can revise its estimates of application performance and implementation cost.

This processor design flow introduces a number of key questions:

- What is specified in a processor configuration?
- What are the measures of processor performance for this specific application?
- What are the cost goals for this processor implementation?
- What kind of cost and performance improvement is feasible for different classes of applications?
- What is the appropriate set of application source code and application data that will drive the processor optimization process?
- How might the application source code change to accommodate processor configuration changes?
- How does automation of processor generation reduce design time and design errors?
- How does the processor fit into the rest of the SOC design?

These key questions are addressed in the course of this book. Chapter 4 uses Tensilica's processor generator and the Tensilica Instruction Extension (TIE) language to illustrate how processors can be configured and extended. These tools form the backbone of a methodology that includes concrete metrics of gate count, die area, power dissipation, clock frequency, and application cycle count as primary measures of performance and cost. Various processor configurations dictate different changes in application source code—ranging from no change to significant exploitation of new application-directed datatypes and operators.

The process pictured in Fig 3-1 also highlights the opportunity for automatic processor optimization. Chapter 4 discusses automatic processor-optimization technology, based on advanced compilers, that identifies application hot spots and candidate instruction sets, evaluates the impact on both application performance and processor hardware cost, selects a Pareto-optimal solution, and generates both an optimized hardware design and software tools for the chosen processor configuration. The role of these compiler-based methods is to automate the path shown in bold in Fig 3-1, allowing thousands of candidate architectures to be evaluated in minutes.

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.

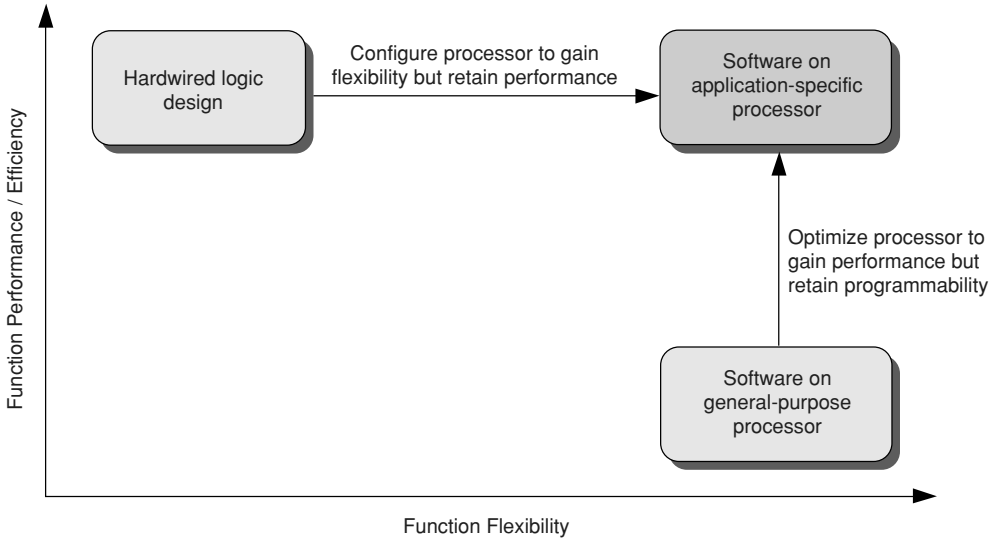

Figure 3-2 Migration from hardwired logic and general-purpose processors.

SOC designers face a major dilemma as they architect and implement a system. Many subsystems 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 100 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.

By contrast, processors show their superior flexibility when function changes are required. Modifying and retesting a software change may be more than 100 times faster and cheaper than modifying the circuits on a large integrated circuit. This contrast is shown in Fig 3-2. 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. Fig 3-2 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 design-

ing 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. Second, 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.

3.2 Accelerating Processors for Traditional Software Tasks

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.

3.2.1 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 bitwise 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 datatypes using relatively long sequences of primitive integer operations. For example Chapter 5, 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.

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.

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.

3.2.2 Explaining 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 handcrafted software tools. The general-purpose processor represents a compromise where features that provide modest benefits to all customers supersede features that provide dramatic benefits to a few. This design compromise is necessary because the historic costs and difficulty of manual processor design mandate that only a few different designs can be built. 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.

3.2.3 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 Fig 3-3.

A block diagram for a configurable processor is shown in Fig 3-4, again using Tensilica's Xtensa processor as an example. The figure identifies baseline instruction-set architecture features, scalable 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 and type of execution units, the num-

ber 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).

Instruction Set	• 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)
Memory System	• 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, multiported memories) into the address space of the processor • Slave access to local memories by external processors and DMA engines
Interface	• 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
Processor Peripherals	• Timers • Interrupt controller: number, priority, type, fast switching registers • Exception vectors addresses • Remote debug and breakpoint controls

Figure 3-3 Processor configuration and extension types.

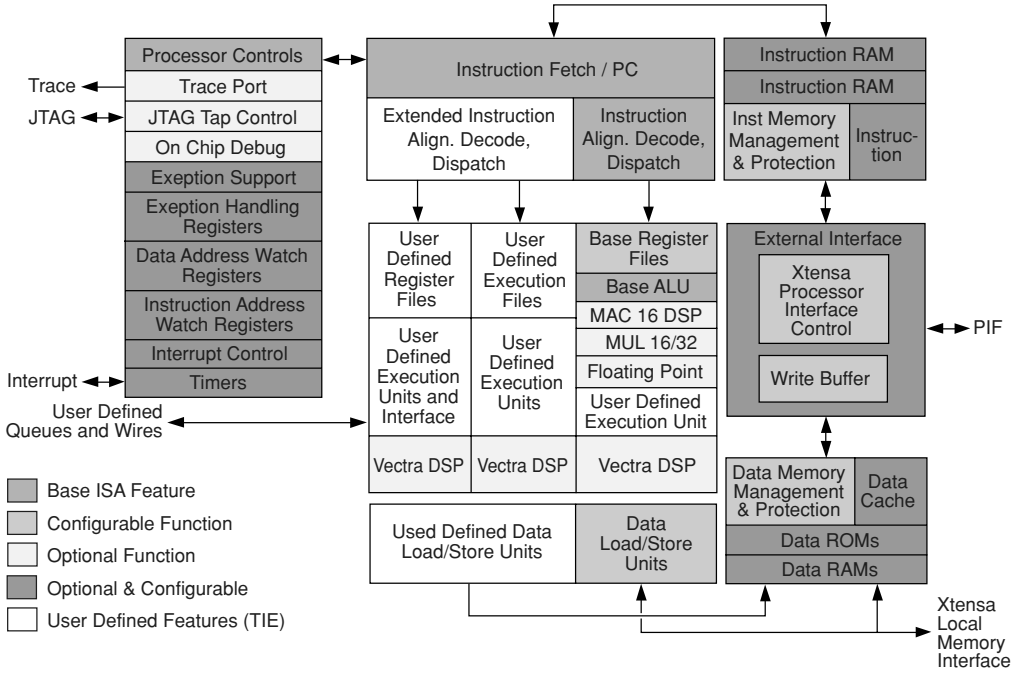

Figure 3-4 Block diagram of configurable Xtensa processor.

3.2.4 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—and 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 while preventing the creation of instructions with subtle bugs that could prevent the basic processor from functioning correctly.

A simple example written in the TIE language illustrates the potential simplicity of this approach. Fig 3-5 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 AR register file, multiplies together the corresponding 8-bit values, and sums adjacent 16-bit multiplication results into a

pair of 16-bit accumulated 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 3-5 Simple 4-way MAC TIE example.

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

- New data-path elements, including four 8 x 8 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.

3.2.5 Memory Systems and Configurability

Even today, on-chip memories consume more silicon area than any other single type of circuit. Increases in data bandwidth, data-encoding complexity, and data resolution are all likely to increase the fraction of chip area consumed by memory. As a result, efficient use of memories becomes even more important over time.

The data-intensive role for configurable processors further emphasizes the criticality of memory systems and interfaces. Six key characteristics of processor memory systems are addressed by memory-system configurability:

Local bandwidth: The performance of many embedded applications (especially in signal, network, and media processing) are limited by the basic data rate of transfers between processor

memories and processor-execution units. By extending the fundamental width for application-specific data words, the processor can transfer data at higher speeds—typically 8 or 16 bytes per cycle—between processor registers and execution units and local data RAMs and caches. Moreover, combining data operations with load/store operations in a single instruction often doubles the effective local processing bandwidth.

Interface bandwidth: For many data-intensive applications, high local-memory bandwidth is necessary but not sufficient. The interface between the processor and the rest of the system must also carry high-bandwidth data streams. Increased interface bandwidth is particularly important as the processor's computational bandwidth increases. The extensibility of the interface for wide data words, fast block transfers, high-speed DMA operations, and application-specific direct data interfaces all contribute to useful interface bandwidth that can easily exceed 10x that of 32-bit, general-purpose RISC processors.

Latency: Many applications have tight interaction between the fetch and use of data. Data-access latency, not just the overall data bandwidth, may be decisive in overall application throughput. Single-cycle access to local memories, and even to surrounding logic, enables an application to generate an address, load new data, and act on the data in a single instruction over just two or three pipeline stages.

Memory scalability: Pipelined processors depend on the ability to tightly couple instruction and data memories into the microarchitecture. Different applications have vastly different memory requirements, yet the configurable processor design must scale across a wide spectrum of memory sizes and types. The ability to scale from local pipelined memories as small as 1 Kbyte to up into hundreds of Kbytes, and to make arbitrary combinations of RAM, ROM, caches, and specialized memories (queues, logic registers), expands the processor's usefulness and guarantees high SOC clock rates even with complex memories.

Multiple operations: Some applications are less sensitive to the number of bytes transferred than to the rate of unique memory operation completions (the number of loads and stores made from and to multiple memory and non-memory locations). Extensible processors can implement multiple load/store units, multiple memory ports, and additional interfaces to specialized memories and registers. These extended interface ports allow two, three, or more independent memory and data-movement operations per cycle.

Support for concurrency: Small, configurable processors are the natural building block for SOCs, but this approach implies task concurrency across a number of processors. To cooperate on a single application, the processors must communicate data in some form. Often, the most logical and efficient communications mechanism is shared memory. For correctness, however, the SOC must implement some hardware-based mechanism to ensure that multiple processors sharing data have the same view of memory at critical points in their computations. While hardware support for memory synchronization can be implemented outside the processor, architectural support simplifies and unifies SOC development.

3.2.6 The Origins of Configurable Processors

The concept of building new processors quickly to fit specific needs is almost as old as the microprocessor itself. Architects have long recognized that high development costs, particularly for design verification and software infrastructure, forces compromises in processor design. A processor had to be "a jack of all trades, master of none."

Early work in building flexible processor simulators, especially ISP in 1970, in retargetable compilers, especially the GNU C compiler in the late 1980s, and in synthesizable processor in the 1990s, stimulated hope that the cost of developing new processor variants might be driven down dramatically. Research in application-specific instruction processors (ASIPs), especially in Europe in code generation at IMEC, in processor specification at University of Dortmund, in micro-code engines ("transport-triggered architectures") at the Technical University of Delft, and fast simulation at University of Aachen all confirmed the possibility that a fully automated system for generating processors could be developed. The VLIW processor work at Hewlett-Packard has also produced forms of processor configurability, found in both the PICO (Program-In-Chip-Out) and LX projects of HP Labs. However, none of these projects has demonstrated a complete system for microprocessor hardware design and software environment generation (compiler, simulator, debugger) from a common processor description. (Note: The last section of this chapter suggests further reading on this early research work.)

While all of this work contributes to the principles of fully automated processor generation, none constitutes a full or automated processor-generation system. Automatic processor generators can be characterized as meeting all of the following criteria:

1. Both hardware description and software tools are generated from a single processor description or configuration.
2. Automatically generated software tools include, at a minimum, a C compiler, assembler, debug, simulator, and a runtime environment or bindings for a standard real-time operating system.
3. The automatically generated processors are complete in the sense that all generated processors are guaranteed to support arbitrary applications programs written, for example, in C. Put another way, all of the automatically generated processors are capable of running programs. They are not merely sequencers.
4. The processor configuration or description is significantly more abstract than an HDL description for an equivalent processor, so hardware and software engineers without deep knowledge of processor architecture can nevertheless understand and develop new processors.
5. The processor-generation process is automated: software programs alone generate detailed processor implementations from the processor description or configuration, without human intervention, in minutes or hours.

Tensilica's processor generator is not the only commercial product in this domain, though

arguably it's the most complete. ARC International's ARCTangent processors and ARChitect tool provide a form of processor extension based on RTL modification without direct hooks to software tools. MIPS Technologies' "User Defined Instructions," supported by the company's Pro series of processors, also provides some means for structured additions to the processor hardware design.

3.3 Example: Tensilica Xtensa Processors for EEMBC Benchmarks

To assess the impact of configurable processing, we must look at processor performance over a variety of applications. Public performance information on large embedded applications for a wide range of processors is scarce. We therefore turn to the best available benchmark data to prove that configurable processors deliver significant performance and efficiency benefits compared to general-purpose processors over a wide range of embedded applications. No single benchmark can accurately capture the full diversity and wide range of embedded applications. Consequently, *EDN* magazine sponsored the creation of a comprehensive embedded benchmark suite that would be more informative than the simple synthetic Dhrystone benchmark once used to rate embedded processor performance. Since 1997, the resulting organization—dubbed EEMBC for EDN Embedded Microprocessor Benchmark Consortium (and pronounced "embassy")—has attracted participation by more than 40 leading processor and software companies. Working together, these companies developed both a set of benchmarks and a fair process for running, measuring, certifying, and publishing test results.

These benchmarks cover a wide range of embedded tasks, but the bulk of the certified results are available for three suites: consumer, telecommunications, and networking. EEMBC rules allow reporting of two types of results: "out-of-the-box" and "full fury." The out-of-the-box scores are based on compiling and running unmodified C code on the processor as realized in silicon or using a cycle-accurate processor simulator. Full-fury scores allow modifications to the C code and hand-coding in assembly language. The full-fury optimizations also allow the use of application-specific configurations to demonstrate the abilities of processor-extension technology.

Each EEMBC benchmark suite consists of a number of different programs written in C. Many of the component programs are run with several data sets or parameter values. The performance for each benchmark within the suite is compared to a reference processor, and the performance ratios for all of the components are averaged using a geometric mean—the accepted method of averaging performance ratios. For each of the benchmark suites in the following sections, we show both the benchmark performance per MHz of clock and the full performance at the rated clock speed of the processor. The clock speeds are taken directly from the corresponding vendor specs for the processor.

In all cases, the data in this section is taken directly from the certified results on the EEMBC Web site at *http://www.eembc.org*, as of April 2003. In each case, we compare results for the Xtensa V processor to the fastest processor from each relevant vendor or architecture for which complete results are available. The bulk of the data represents licensable processor-core

architectures, though in a few cases, additional architectures are included in the comparison. The NEC VR4122 implements a 32-bit MIPS instruction set and uses a pipeline and interface roughly similar to that of the Xtensa processor. The MIPS 20Kc implements a 64-bit MIPS instruction set and incorporates a pipeline capable of issuing multiple instructions per clock. The ARM architecture is represented in the EEMBC benchmarks by the ARM1026EJ-S. This processor core implements the 32-bit ARMv5TEJ instruction set architecture and uses a dual 64-bit bus interface.

To demonstrate the impact of extensible processing on each suite, we include the results for a baseline configuration of the Xtensa processor, with no added TIE instructions, running simple compiled C code, and for an optimized configuration, including TIE instructions and optimized code. A different optimization of the Xtensa processor is used for each benchmark category, just as an SOC designer would develop individual configurations for each set of related application tasks on an SOC. We include the best results available for the other processors. For most, only baseline results are available.

3.3.1 EEMBC Consumer Benchmarks

Video processing lays at the heart of many consumer electronics products—in digital video cameras, in digital televisions, and in games. Common video tasks include color-space conversion, 2D filters, and image compression. The EEMBC Consumer benchmarks include a representative sample of all these applications. Fig 3-6 and Fig 3-7 show the performance on EEMBC's Consumer benchmark suite for the target set of processors plus the Philips Trimedia TM1300, a very-long-instruction-word (VLIW) processor that is optimized for multimedia applications.

Optimized EEMBC scores can include both hand-tuning of code and tuning of the processor configuration for this class of consumer tasks. For Xtensa processors, roughly 200,000 gates of added consumer instruction-set features are implemented in TIE. Fig 3-6 shows the Consumer benchmark scores (ConsumerMarks) per MHz, where the STMicroelectronics ST20 at 50MHz sets the performance reference of 1.0. These results show that the configured Xtensa processor provides about three times the cycle efficiency of a good media processor (the Philips Trimedia TM1300), about 33 times the cycle efficiency of a good 64-bit RISC processor (the MIPS 20Kc), and 50 times the performance of a basic 32-bit RISC processor (the NEC VR4122). Fig 3-7 shows the benchmark performance including the impact of clock frequency. These results confirm that even substantial processor optimizations do not significantly reduce clock frequency, so the high instruction/clock efficiency achieved through processor extension translates directly into high absolute performance.

3.3.2 Telecommunications

Telecommunications applications present a different set of challenges. Here the data is often represented as 16-bit fixed-point values, as densely compacted bit-streams or as redundantly encoded channel data. Over the past 10 years, standard digital signal processors (DSPs) have

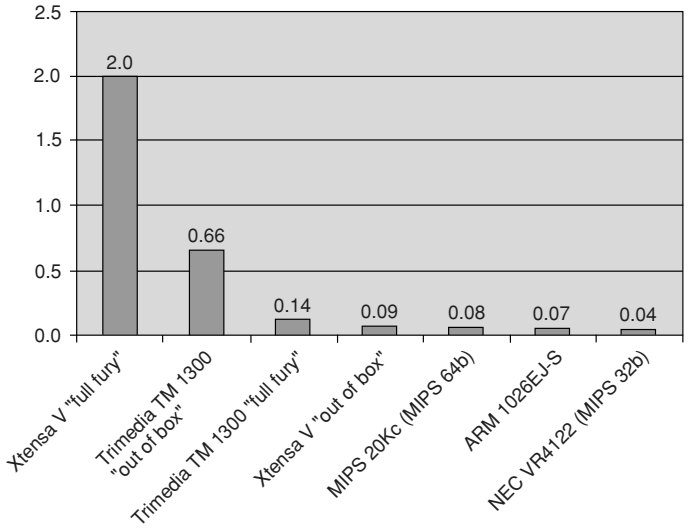

Figure 3-6 EEMBC ConsumerMarks—per MHz.

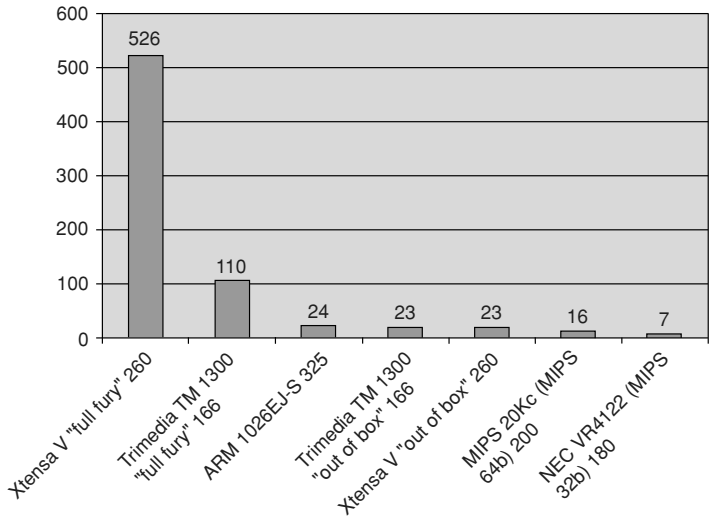

Figure 3-7 EEMBC ConsumerMarks—performance.

evolved to address many filtering, error correction, and transform algorithms. The EEMBC Telecom benchmark includes many of these common tasks.

Fig 3-8 shows the efficiency of the Xtensa processor (including the Vectra V1620-8 vector-coprocessing option) and various architectures, on a performance per MHz basis, where the

IDT 32334 (a 32-bit MIPS architecture) at 100MHz sets the performance reference score of 1.0. The other architectures shown include a high-end DSP (the Texas Instruments TMS320C6203) plus the ARM and MIPS processors. For the “full fury” Telecom scores (TeleMarks). The figure includes results for optimized configurations of the Xtensa V core and hand-tuned code for the TI ’C6203. The Xtensa processor has been specifically optimized for two of the benchmarks in the telecom suite: convolutional coding and bit allocation. Fig 3-9 shows the dramatic impact of application-directed instruction sets and code tuning for each of the optimized DSP architectures relative to conventional 32- and 64-bit processors. The configured processor is 30 times faster than all of the conventional RISC cores.

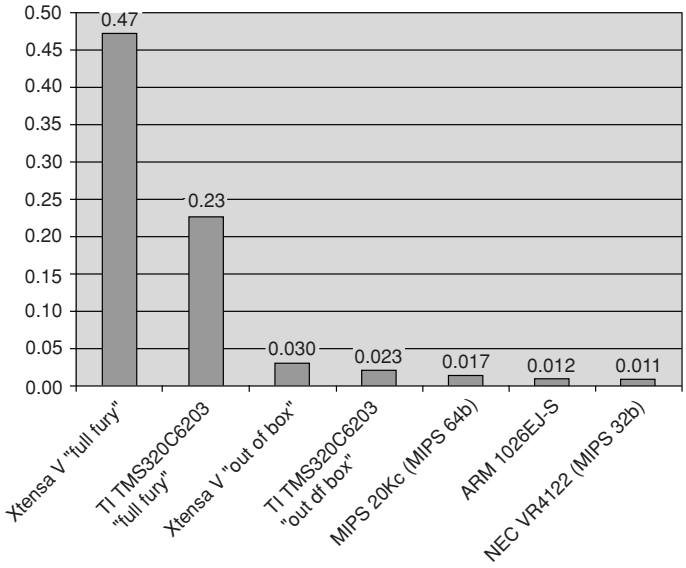

Figure 3-8 EEMBC TeleMarks—per MHz.

3.3.3 EEMBC Networking Benchmarks

Networking applications have significantly different characteristics than consumer and telecommunications applications. They typically involve less arithmetic computation, generally contain less low-level data parallelism, and frequently require rapid control-flow decisions. The EEMBC Networking benchmark suite contains representative code for routing and packet analysis. Fig 3-10 compares EEMBC Network performance (NetMarks) per MHz for Xtensa V processors with and without extensions and several other architectures, where the IDT 32334 (a 32-bit MIPS architecture) at 100MHz sets the performance reference score of 1.0. The optimizations for the Xtensa processor are small, adding less than 14,000 additional gates (less than $0.2mm^2$ in area) to the processor. The extended Xtensa processor achieves about seven times the

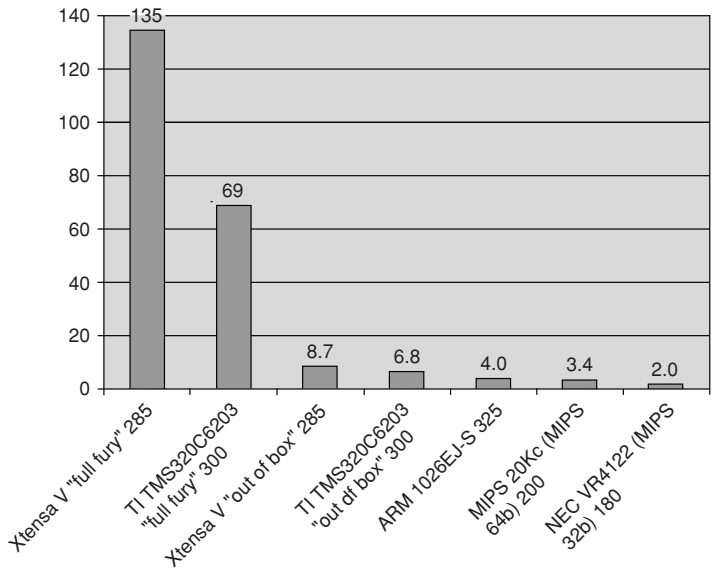

Figure 3-9 EEMBC TeleMarks—Performance

cycle efficiency of a good 64-bit RISC processor core and more than 12 times the efficiency of a 32-bit RISC processor core. All of the processors shown run at generally comparable clock frequencies, but performance varies widely, giving the overall NetMark performance shown in Fig 3-11.

The EEMBC benchmark results show that extensible processors achieve significant improvements in throughput across a wide range of embedded applications, relative to good 32- and 64-bit RISC, DSP, and media processor cores.

Interestingly, when EEMBC certified and released the optimized Xtensa processor benchmark results, some observers complained, "That's not fair—the Xtensa processors have been optimized for the benchmarks." That's exactly the point of configurable processors— in embedded applications, much of the workload is known before the chip is built, so optimizing the processors to the target software is an appropriate strategy. The EEMBC benchmark codes are simple but reasonable stand-ins for types of applications at the heart of real embedded systems. The process of optimizing the processors for the EEMBC benchmarks—all the Xtensa data presented here—was the result one graduate student's summer project and should closely match the real-world process of optimizing processors for SOC applications.

3.3.4 The Processor as RTL Alternative

Extensible processors can play another fundamental SOC role in addition to serving as a faster processor running software tasks. In many cases, the instruction set can incorporate complex logic, bit manipulation, and arithmetic functions that look strikingly like the data paths of hard-

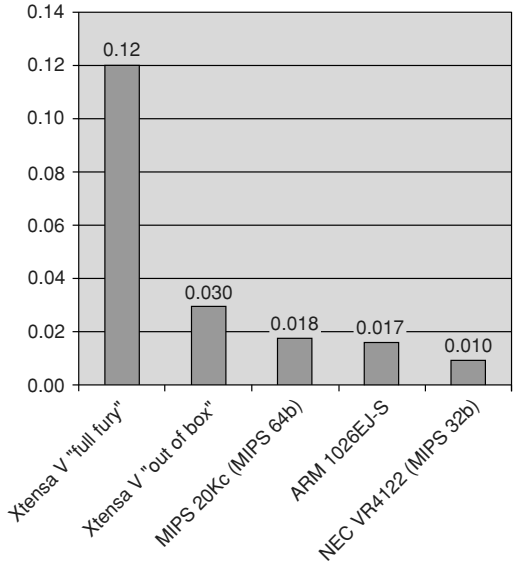

Figure 3-10 EEMBC NetMarks—per MHz.

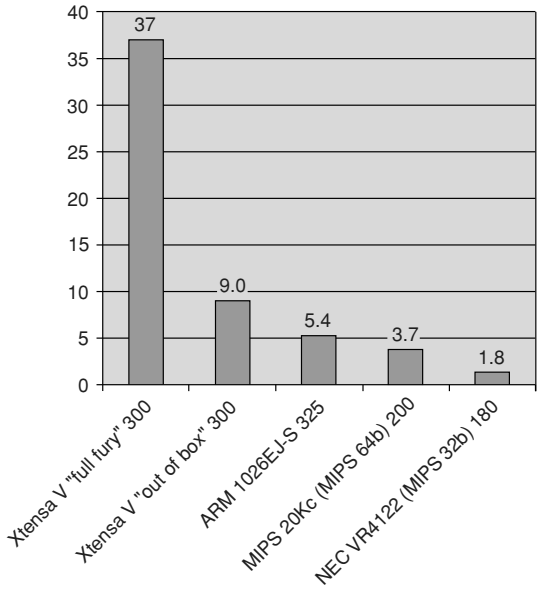

Figure 3-11 EEMBC NetMarks—Performance.

wired logic blocks. Extensible processors are easily configured to have similar throughput, similar interfaces, and similar silicon area and power dissipation characteristics when compared to RTL logic blocks.

The big differences between a dedicated function block implemented as logic and an application-specific processor may boil down to just two. Both turn out to be advantages for the extensible processor. First, the processor structure and generation process make the function easier to specify, model, and debug than the equivalent logic block developed just using RTL. Second, the processor is easily reprogrammed for new tasks using standard programming tools, limited only by the basic functions captured in its application-specific instruction set and configuration.

The opportunity to replace complex RTL blocks with processors depends on some important differences between this class of configurable processor and traditional general-purpose processors. The differentiating attributes of processors used as RTL alternatives include the following:

- Support for wide, deep, application-specific data paths as extended instructions.
- Support for very simple system-programming models without requiring interrupts and exception handing.
- Support for novel memory systems, not just caches, including wide memories, multiported data RAMs, instruction ROM, direct mapping of queues and special registers.
- Support for direct connection of external signals as inputs to instructions and direct connection of instruction results to external output signals.

With these capabilities, the extensible, application-specific processor can efficiently substitute for blocks of RTL. Like RTL-based design, configurable processor technology enables the design of high-speed logic blocks tailored to the assigned task. The key difference is that RTL state machines are realized in hardware and logic blocks based on extensible processors realize their state machines with firmware.

Hardwired RTL design has many attractive characteristics—small area, low power, and high throughput. However, the liabilities of RTL (difficult design, slow verification, and poor scalability for complex problems) are starting to dominate as SOCs get bigger. A design methodology that retains most of the efficiency benefits of RTL but with reduced design time and risk has a natural appeal. Application-specific processors as a replacement for complex RTL functions fit this need.

An application-specific processor can implement data path operations that closely match those of RTL functions. The equivalent of RTL data paths are implemented using the integer pipeline of the base-processor, plus additional execution units, registers, and other functions added by the chip architect for a specific application. A high-level instruction-set description is much more concise than an RTL description because it omits all sequential logic, including state-machine descriptions, pipeline registers, and initialization sequences.

The new processor instruction definitions are available to the firmware programmer via

the same compiler and assembler that serve the processor's base instructions and register set. All sequencing of operations within the processor's data paths is controlled by firmware through the processor's existing instruction-fetch, decode, and execution mechanisms. Firmware for these application-specific processors can be written either in assembly code or in a high-level language such as C or C++.

Extended processors used as RTL-block replacements routinely use the same high-throughput techniques as traditional data-path-intensive RTL blocks: deep pipelines, parallel execution units, problem-specific state registers, and wide data paths to local and global memories. These extended processors can sustain the same high computation throughput and support the same low-level data interfaces as typical RTL designs.

However, control of the extended-processor data paths is very different from the control functions implemented in RTL-based designs. Cycle-by-cycle control of the processor's data paths is not fixed in hardwired state transitions. Instead, the sequence of operations is explicit in the firmware executed by the processor. Compare Fig 3-12 with Fig 2-2 in the previous chapter. In processor-based design, control-flow decisions are made explicitly in branches, and memory references are explicit in load and store operations, computation sequences are explicit sequences of general-purpose and application-specific processor instructions.

Implementing a complex logic function using a processor provides important new structure for both the design process and the final design. The essential, defining characteristics of the function are embodied in the processor's extended instruction set.

The transitory details become the program that runs on the processor. This separation aids rapid discovery of a natural and efficient compromise between the seemingly incompatible goals of narrow tuning to the exact needs of the application and sufficient flexibility to accommodate changing needs.

Using processor generation as an alternative to RTL design also brings to bear a powerful set of software-related tools and methodologies. The essential function description is a behavioral C or C++ program rather than a structural logic description in a Verilog or VHDL file. Processor simulation is orders of magnitude faster compared to RTL simulation and waveform viewers. The full spectrum of software-development tools, from RTOSes and standard interconnect buses to software project management and source-level debuggers, all map directly from the traditional control-processor world to this new domain of processor-as-logic-replacement.

Thus, easy and complete generation of new processors and tools creates two distinct benefits—higher performance and efficiency for processors in traditional processor roles and easier design and reprogrammability for processors in RTL-alternative roles. Additional benefits come from using the same class of processors and tools for both processor and hardwired logic roles. For many complex system problems, all major logic functions can be implemented with processors. Using processors to implement logic blocks means that all simulation can be done with fast processor simulators; all functions can be described at a high level (as C/C++ programs and processor configurations); debugging can be performed with C/C++ source-level; all memories are managed, and potentially shared by software; all functions can be continuously updated

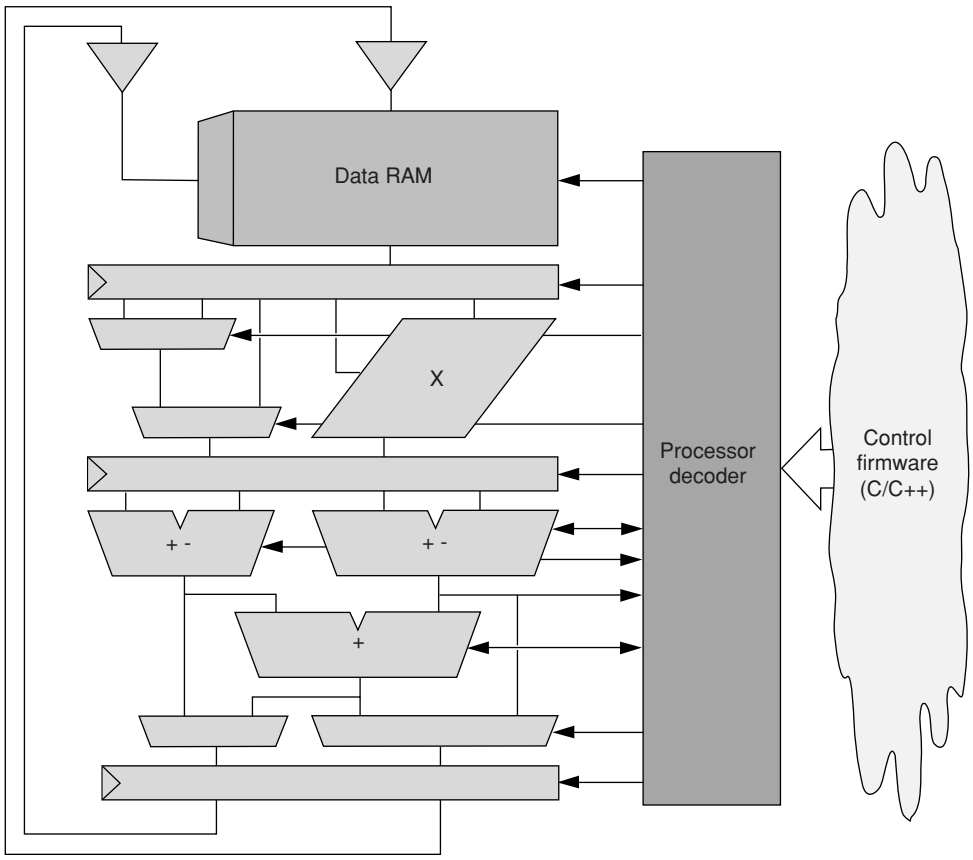

Figure 3-12 Configurable processor as RTL alternative.

just by dynamically changing the software; and the entire design is commonly portable across silicon fabrication processes and easily reusable in parts or in whole. This unification makes the design of subsystems, and the system as a whole, quicker and less costly.

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 in parallel, to more efficiently implement the system's functions. The designer can exploit concurrency 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, bitwise 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 basic operations 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 nondependent 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.

  That operation-level concurrency can often be exploited by applying the same operator to a number of different operands at the same time for single-instruction multiple data (SIMD) parallelism. Additional operation-level concurrency can be exploited by executing several independent operations at the same time, using a very-long-instruction word (VLIW) processor architecture or superscalar processor implementation. Pipelined implementation of execution units also increases realized concurrency – combinations of dependent operations can form pipelined execution units, while independent sequences of operations flow down the pipeline in parallel with one another.
- **Task-level parallelism:** 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—for example, 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 digital CMOS circuits). If the function implemented in the circuit can be scaled through parallel logic implementation by some factor s(>1), so that the circuit has more transistors (for more parallelism) by a factor of s, but can achieve the same throughput at frequency reduced by s we can reduce the operating voltage by roughly s as well, so we see a significant reduction in power:

$$P_{new} \propto (sC)\left(\frac{V}{s}\right)^2\left(\frac{f}{s}\right) = \frac{P_{original}}{s^2}$$

In practice, s 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 s. On the other hand, the scaling of switched capacitance (sC) with performance may be unrealistically conservative. As shown in the EEMBC benchmarks cited earlier, small additions to the processor hardware, which slightly increase switched capacitance, often lead to large improvements in throughput. This simplified analysis underscores the

important trend toward increased parallelism in circuits.

3.4.3 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 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. Chapter 4 discusses the refinement method in greater depth.

3.4.4 Forms of Partitioning

When we partition a system's functions into multiple interacting function blocks, we find several possible forms or structures. A quick overview of these basic types illustrates a few key partitioning issues.

- **Heterogeneous tasks:** As described above, many systems contain distinct, loosely cou-

pled subsystems (e.g., video, audio, networking). These subsystems share only modest amounts of common data or control information and can be implemented largely independently of each other. The chief system-level design concern will be supplying adequate resources for the sum of all the requirements of the individual subsystems. Likely needs include memory capacity, bus bandwidth, and access from a system-control processor. In many cases, the required capacity can come from a common pool of on-chip resources to simplify design and encourage flexible sharing, especially if the resource needs are uncertain. Fig 3-13 shows one plausible topology for such a system. This system design assumes that networking, video, and audio processing tasks are implemented in separate processors, sharing common memory, bus, and I/O resources.

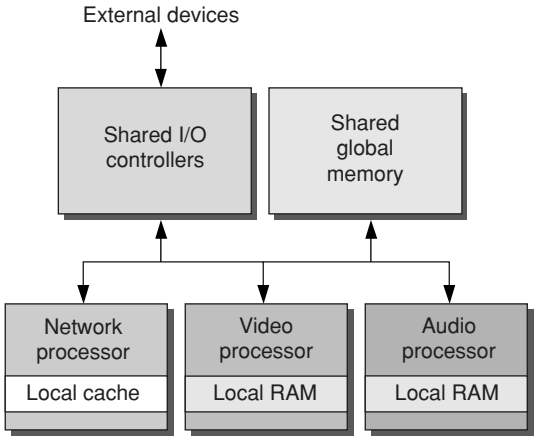

Figure 3-13 Simple heterogeneous system partitioning.

- **Parallel tasks:** Some embedded tasks are naturally and explicitly parallel. Communications infrastructure equipment, for example, often supports large numbers of wired communications ports, voice channels, or wireless frequency-band controllers. These tasks are easily decomposed into a number of identical subsystems, perhaps with some setup and management from a controller, as shown in Fig 3-14. These parallel processors may share common resources as long as the shared resource doesn't become a bottleneck. Even when the parallelism is not obvious, many system applications still lend themselves to parallel implementation. For example, an image-processing system may operate on a dependent series of frames, but the operations on one part of a frame may be largely independent of operations on another part of that same frame. Creating a two-dimensional array of subimage processors may achieve high parallelism without substantial algorithm redesign.
- **Pipelined tasks:** Some embedded system functions may require a long series of dependent operations that precludes use of a parallel array of processors as discussed above. Nevertheless, the algorithms can be naturally organized into phases, such that one phase of

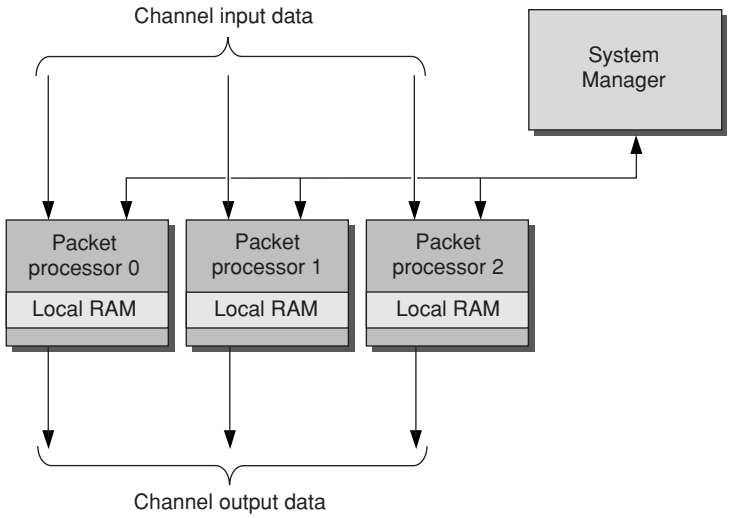

Figure 3-14 Parallel Task System Partitioning

the algorithm can be performed on one block of data while a subsequent phase is performed on an earlier block. (This arrangement is called a systolic-processing array.) Data blocks move down the pipeline of processors, and each processor in the pipeline is dedicated and optimized for a particular subsequence of the entire function.

Fig 3-15 shows an example of a systolic array for decoding compressed video. The Huffman decode processor pulls an encoded video stream out of memory, expands it, and passes the data through a dedicated queue to the inverse discrete cosine transform (iDCT) processor, which performs a complex sequence of tasks on the image block and passes that block through a second queue to a motion-compensation processor, which combines the data with previous image data to produce the final decoded video stream.

- **Hybrids:** The above three system-partitioning cases are unrealistically simple. Real systems often require a rich mixture of these partitioning styles. For example, one of the independent subsystems of Fig 3-13 might be best implemented in a group of parallel processors. Or perhaps the system manager in Fig 3-14 might be implemented by another processor as supervisor. The nature of the application and the ingenuity of the system architect drive the parallel system structure.

Complex systems may generate correspondingly complex topologies. The combination of optimizing the architecture of individual processors and optimizing system organization, including use of parallel processors, provides a new rich palette of possibilities for the system designer.

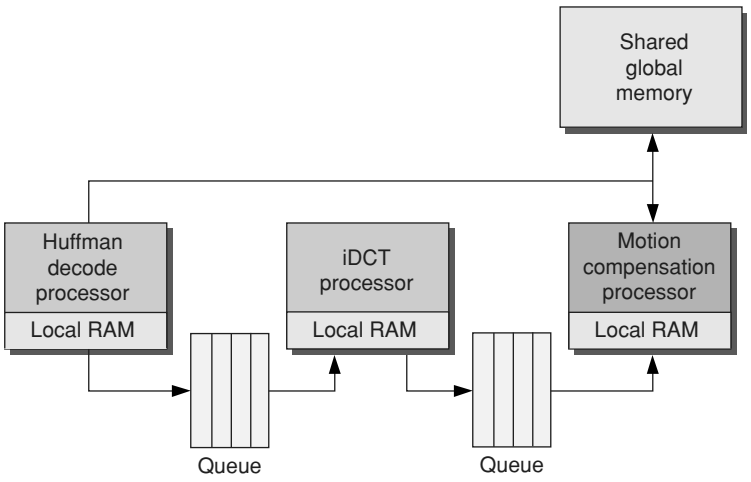

Figure 3-15 Pipelined task system partitioning.

3.4.5 Processor Interface and Interconnect

Full exploitation of the potential for parallel processors depends strongly on the characteristics of the processor's hardware interface and programming support for communication. Different applications make different demands on communication between blocks. Four questions capture the most essential system-communications performance issues:

1. **Required bandwidth:** To sustain the required throughput of a function block, what sustained input data and output data bandwidths are necessary?
2. **Sensitivity to latency:** What response latency (average and worst-case) is required for a functional block's requests on other memory or logic functions?
3. **Data granularity:** What is the typical size of a request—a large data block or a single word?
4. **Blocking or nonblocking communications:** Can the computation be organized so that the function block can make a request and then proceed with other work without waiting for the response to the request?

For extensible processors to be effective in a broad range of roles, replacing both traditional processors and traditional hardwired logic blocks, they must provide a basic vocabulary of communications interfaces that suit applications across these four dimensions. Fig 3-16 highlights the three basic forms of interface natural to processors tuned for SOC applications:

1. Memory-mapped, word-sized interface—typically implemented as a local-memory-like connection.
2. Memory-mapped, block-sized connection—typically implemented as a bus connection.
3. Instruction-mapped, arbitrary-sized connection—typically implemented as a direct point-to-point connection. Instruction-mapped connections can range from a single bit to thousands of bits.

Note that traditional processor cores, which evolved from standalone processors, typically provide only the block-oriented, general-bus interface. Low bandwidth, long latency, or inappropriate interface semantics would either degrade performance or complicate development for parallel applications. Each of these interfaces has particular strengths, as outlined in Fig 3-17.

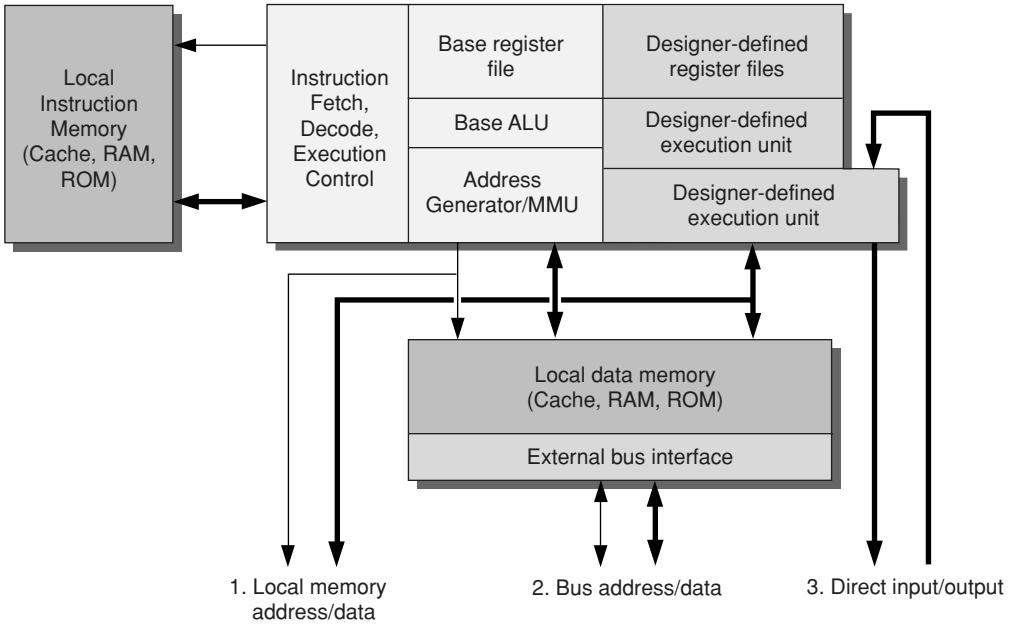

Figure 3-16 Basic extensible processor interfaces.

Interface Type	Interface Characteristics	Interface Uses
Local memory interface	Memory mapped, word oriented, fixed latency	• Word access to shared local memory • Word access to memory-mapped mailbox register or data queue
Bus interface	Memory mapped, block-data oriented, variable latency	• Block access to shared remote memory area • Bus access to another processor's local memory
Direct port interface	Instruction mapped, word oriented, dedicated connection	• Control and status input and output • Data operand transfers

Figure 3-17 Interface characteristics and uses.

3.4.5.1 Bus Interface

General bus interfaces are quite familiar to designers of microprocessor-based systems. These interfaces support a wide range of memory-mapped read and write operations, including both word transfers and block transfers for cache-line fills and dirty-cache-line write-backs. A processor typically uses this interface to access system wide shared resources such as main memory, I/O device-control registers, and other hardware functions. Because there is often contention for these shared resources—either directly at the resource or for any shared bus that connects these resources to the processor—the interface must support variable or indeterminate latency. Often, this interface will support split transactions (separation of data request and data response), bus errors (signaling to the processor that the request cannot be correctly satisfied), and transactions in support of cache coherency (hardware support for maintaining the illusion to software that each memory address holds exactly one value).

Normally, the memory and registers addressed over the bus are separate from the processors that access them. With a more sophisticated bidirectional multimaster bus interface, even the local resources of one processor become visible to other processors on the bus. A processor's private local data RAM, for example, can be made public and available as a source and destination of both I/O-device-to-processor traffic (direct memory access, or DMA) and processor-to-processor data traffic.

3.4.5.2 Local-Memory Interface

Memory hierarchies play an important role in high-performance embedded processors. The big performance gap between large, shared, off-chip memories and small, dedicated, on-chip memories makes it natural for processor pipelines to require that the first level of the instruction and data storage is closely coupled to the processor and has high bandwidth and a latency of roughly one processor clock cycle. These local memories may be structured as caches (data movement from off chip is automatic), or as local memory spaces (data movement from off chip is directed explicitly by software).

Even though these memories are accessed through optimized private interfaces, these interfaces can be extended to support other high-bandwidth, low-latency, memory-mapped functions. A local interface can support the same range of data-word sizes as the private memory (8, 16, 32, possibly 64 and 128 bits, or wider). This local-memory interface assumes the same quick access time (typically one cycle) as the private memory, but often includes some stall mechanism to allow arbitrated access to a shared resource.

This high speed makes a local interface suitable for functions where high-bandwidth and low-latency data access is required. In some cases, random-access shared memories are connected to this local-memory interface. In other cases, the local-memory interface will be used to connect to a queue, the head or tail of which is mapped into a particular memory address. In some cases, memory-mapped system registers representing the control or data interface of some external function are connected to the local-memory interface.

3.4.5.3 Direct Connection

The unique characteristics of extensible processors and on-chip interconnect create the opportunity for third, fundamental form of interface—direct port connections. This interface exploits two phenomena. First, the intrinsic on-chip bandwidth of deep submicron semiconductor devices is surprisingly huge. For example, in a six-layer-metal process, we might expect that two layers will be available in the X dimension and two layers in the Y dimension for global wiring. In 90nm technology, we can expect 500MHz processor clock frequencies and roughly four metal lines per micron per metal layer.

Therefore, the theoretical bandwidth at the cross-section of a 10mm x 10mm die is roughly:

$$\mathrm{BW}_{\text{cross-section}} = (500 \bullet 10^6\,\mathrm{Hz}) \times (4\ \text{lines}/\mu\mathrm{m}) \times (2\ \text{layers}) \times (10\mathrm{mm}) \times (10^3\,\mu\mathrm{m/mm})$$

$$= 4 \bullet 10^{13}\ \text{bits/sec}$$

$$= 40\ \text{terabits/sec (Tbps)}$$

This high theoretical bandwidth highlights the potential for parallel on-chip processing, even for applications with very high bandwidth demands between processing elements. Note that 40 Tbps is much greater than the bus bandwidth or internal operand bandwidth of any conventional, general-purpose 32-bit or 64-bit processor core. The fact that conventional processors use only a small fraction of this available bandwidth motivates the search for alternatives that can better exploit this potential.

Second, extensible processors support wide, high-throughput computational data paths that vastly increase the potential performance per processor. These processors can be configured with pipelined data paths that are *hundreds* or, in extreme cases, *thousands* of bits wide. Consequently, each data path can potentially produce or consume several operand values per cycle, so the total data appetite for a single extensible processor can scale to literally terabits per second. If only a fraction of this bandwidth is consumed by processor-to-processor communications, only a few dozen processors are required to use the available 40 Tbps of cross-section bandwidth. It's therefore feasible to exploit the unique bandwidth characteristics of advanced semiconductor technology using multiple processors with high-bandwidth interfaces.

To best exploit the available bandwidth, direct processor-to-processor connections are not mapped into the address space of the processor. Instead, processors send or receive these data values as a byproduct of executing particular instructions. Such communications are *instruction-mapped* rather than memory-mapped because the processor-to-processor communications are a consequence of executing an instruction that is not an explicit load or store instruction. Conventional processors use inputs from processor registers and memory and produce outputs in processor registers and memory. Direct processor-to-processor connections allow a third category of source or destination, potentially used in combination with register- and memory-based sources and destinations. Four important potential advantages characterize direct connections versus memory-mapped connections:

- The operational use of the direct-connection interface is implicit in the instruction so, additional instruction bits are not required to explicitly specify the target address of the transfer.
- Neither an address calculation nor a memory load/store port is required, saving precious address and memory bandwidth.
- Transfer sizes are not limited to the normal power-of-two memory word widths, allowing both much larger and application-specific transfers.
- Because transfers are implicit in instruction execution, an unlimited number of parallel transfers can be associated with a single instruction.

Two forms of direct-connect interface—operand queues and control signals—are discussed in detail in Chapter 5, Section 5.6. This expanded vocabulary of basic interface structures offers two fundamental capabilities to the system architect:

1. Richer, wider, and faster interfaces among small processors, which allow higher bandwidth and lower latency among groups of processors on a chip.
2. The combination of both flexible memory-mapped and flexible instruction-mapped interfaces makes a broader range of communications functions directly available to the software.

Together these interface structures create a basic, adaptable computing fabric that serves as a foundation for small, fast, programmable SOC designs.

3.4.6 Communications between Tasks

To effectively use multiple processors, software developers must write programs that communicate appropriately. Several basic modes of communication programming have emerged to handle a variety of circumstances. In some cases, the choice of programming model depends on the data-flow structure among the tasks that comprise the total application. In other cases, the choice depends on the underlying hardware used to implement the function. There are three basic styles of communication between a task producing data on one processor and a task consuming that data on another processor: shared memory, device drivers, and message passing.

3.5 New Essentials of SOC Design Methodology

We have now covered all the basic pieces of the advanced SOC design methodology:

- How extensible processors accelerate traditional software-based tasks.
- How extensible processors flexibly implement hardware-based tasks.
- How to connect multiple processors together to create very sophisticated SOCs.

The emergence of multiple-processor SOCs using configurable and extensible processors as basic building blocks significantly affects three dimensions of the SOC design flow:

1. **Partitioning:** In the traditional SOC-design methodology, architects use two types of functional building blocks: processors with modest performance running complex algorithms in software and hardware with modest complexity performing tasks at high speed. The new design vocabulary of extensible processors gives designers more flexible choices, allowing highly complex functions to be efficiently implemented for the first time. Moreover, the generation of new processor variants is so fast that architects can change their partitioning easily in response to new discoveries about overall system design efficiencies and new market requirements. Greater partitioning choice and greater repartitioning ease significantly expand SOC design horizons.
2. **Modeling:** When the core of the SOC design consists of both RTL blocks and processors, much of the modeling must be done at the lowest-common-denominator level of RTL simulation. In the absence of special hardware accelerators, software-based RTL simulation of

a large subsystem may only achieve hundreds of cycles per second. A complex cluster of processor and RTL logic might run at simulation speeds of only a few tens of cycles per second. By contrast, a fully cycle-accurate extensible-processor instruction-set simulator runs at hundreds of thousands of cycles per second.

A simulation of a complex processor cluster of would routinely sustain tens of thousands of cycles per second on such an instruction-set simulator. This three-order-of-magnitude performance gap means that SOC design teams can run entire software applications to verify their system designs instead of being restricted to running some simple verification tests on a slow HDL simulator. In addition, processor-centric design also makes it far easier to convert early, abstract system-level models written in C or C++ into the final SOC system implementations. Architects can directly evolve early system-level behavioral models into final implementation of task software and configured processors.

3. **Convergence to implementation:** In the traditional SOC design flow, growth in design complexity can severely degrade progress toward a final verified implementation. A small change in requirements can cause major redesign of RTL in one block. Changes to one block often propagate to adjacent blocks and throughout the design. Each block change requires reverification and potential reassessment of the cost, performance, and power consumption of the entire set of blocks. Consequently, much iteration is sometimes required to restabilize the system design after a supposedly "simple" change.

   When extensible processors are used as a basic building block, changes are more easily tried and verified via software. The interface methods are more structured and more rapidly modeled, reducing long redesign cycles. Even if significant repartitioning is required, design elements in the form of software algorithms and processor configurations can be migrated transparently from one processor block to the next. In addition, all the building blocks, whether used as control processors or RTL-replacement engines, are generated in the same portable form from the same tool set. This approach brings uniformity to the VLSI design processor and accelerates floor planning, synthesis, cell placement and routing, signal integrity, design-rule checking, and tape-out. Similarly, with all software built for a single family of compatible processors under a common development environment, development, testing, and packaging of the embedded software suite is unified and simplified.

3.5.1 SOC Design Flow

Compare the advanced SOC design flow in Fig 3-18 to the more traditional SOC flow sketched in Fig 3-14. The fundamental inputs to the flow are SOC interface and functional requirements. The essential input/output interface specification includes physical layer electrical specifications and higher-level logical specifications, including protocol conformance. The essential computational requirements include both specification of the algorithms that relate inputs to outputs and the temporal constraints on algorithmic behavior, including maximum latency, minimum bandwidth, and other "quality of service" characteristics.

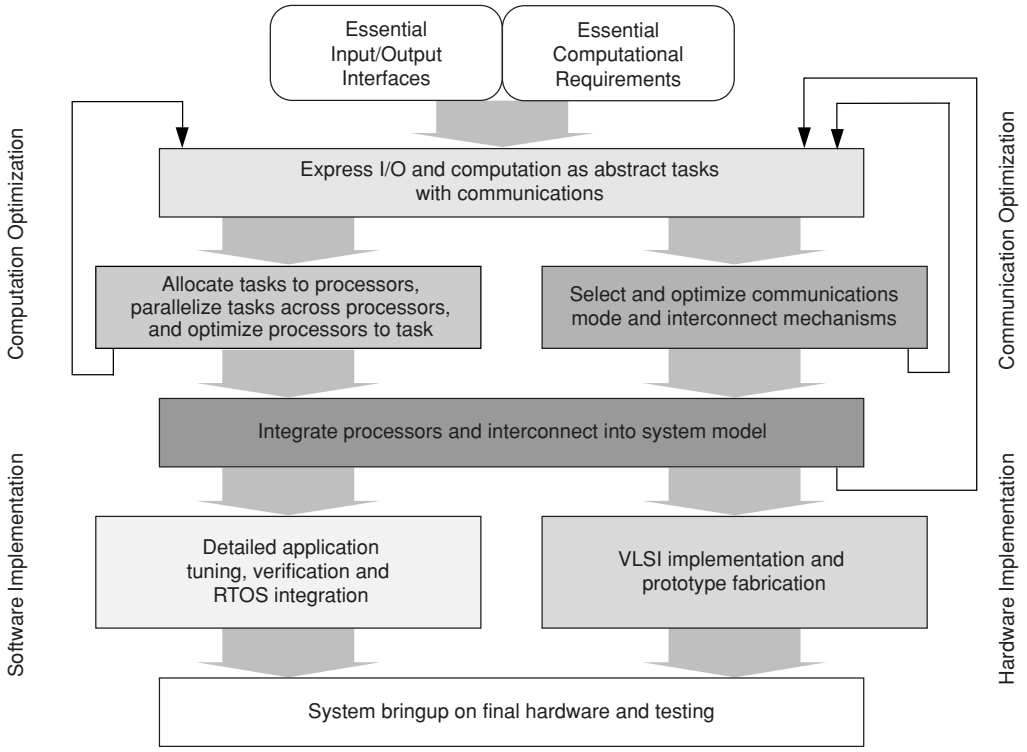

Figure 3-18 Advanced SOC design process.

Within the overall flow are four major subflows: computation optimization, communication optimization, software implementation, and hardware implementation. Almost all feedback loops are found within these blocks (the details are discussed in Chapter 4). Optimization of computation involves finding the right number and type of process (and where necessary, other blocks) to implement the complex nest of computational requirements of the system. Communication optimization focuses on providing the necessary communication software programming model and hardware interconnects among processors and to memories and input/output interfaces, to enable low-cost, low-latency communications, especially for the most critical dataflows in the system. This flow relies on two important principles. First, early system-level simulation is a key tool for creating detailed designer insight into function and performance issues for a complex system architecture. Second, rapid incremental refinement from initial implementation guesses into final application, processor configuration, memory, interconnect, and input/output implementations lets the design team learn from cost and performance sur-

prises and take advantage of new insights quickly. Chapter 4 outlines the issues and describes the recommended system design flow in detail.

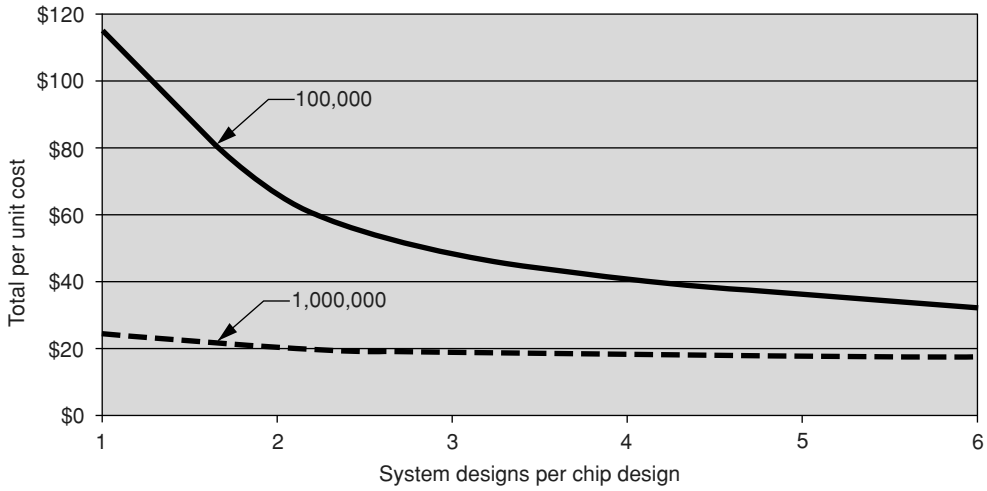

Figure 3-19 Example amortized chip costs (100K and 1M system volumes).

Crucially, this design flow does not make new demands on total design resources or mandate new fundamental engineering skills. Experience with a large number of design teams suggests that both traditional RTL designers and embedded software developers can be comfortable and effective in the design of new processor configurations and application-specific instruction sets using processor-generation tools. Familiarity with C and Verilog are both useful foundations for effective and efficient use of the processor-centric SOC design method. The flow closely depends on the expanded role of configurable processors. Availability of these more flexible SOC building blocks mean fewer design bugs, fewer hardware design iterations, wider exploration of block-partitioning options, earlier block software testing and validation, and in all likelihood, earlier product availability.

3.5.2 The Essential Phases of the New Flow

Every complex SOC design project is different, but the new SOC design flow shown in Fig 3-18 highlights five basic process steps. The SOC design process is intrinsically interdependent, so design teams may perform some of these steps in parallel and may iterate back to an early step after making key observations in the course of later design steps. Nevertheless, this design flow leverages the inherent flexibility of the extensible processor building block and the inherent development productivity of a software-centric design method:

1. **Develop a functional model of system essentials**—Starting from a reference software implementation of key tasks or from an algorithmic description of mission-critical functions, develop a C or C++ program that captures the most performance-demanding or most complex set of system activities. In practice, the architect typically understands that many of the system's functions are quite generic and make few unique demands on processor features, I/O capability, interconnect bandwidth, or memory resources. Only those features expected to drive key design decisions need be modeled at this stage.
2. **Partition the system into multiple processors**—Identify obvious task-level parallelism in the underlying system structure. Decompose tasks with multiple phases into separate pipelined tasks with data communications between tasks. Subdivide each task into parallel computations, where each processor works on a subset of the datastreams in the system. Compile and run the core task or tasks on the baseline processor to establish a reference performance level and to identify major computational bottlenecks. Refine the system model using multiple processors running various software tasks with communications among the processors.
3. **Optimize each processor for its respective task**—For each processor, profile the task performance and identify hotspots within each task. Configure each processor's memories, system peripherals, and interfaces and describe or select instruction-set extensions. Update area and power estimates. Regenerate the software tools and simulators using the new processor extensions. Update applications to utilize more efficient data types and application-specific functions. Refine the processor configurations until the system design meets the per-task cost, power, and performance goals.
4. **Verify system-level cost, performance, and function**—Reintegrate each of the processor models and optimized tasks into the system-level model. Confirm that major partitioning and architecture choices are appropriate. Simulate the system to measure key performance characteristics. Check adequacy of shared resources (memory capacity, interconnect bandwidth, total silicon area, and power budget). Deploy software-development tools and the system model to final application developers.
5. **Finalize the implementation**—Use the final RTL for each processor, the generated memories, and other logic to floor plan, synthesize, place, and route each block. Stitch together global clocks, debug and test scan-chain hardware and the power infrastructure. Run full-chip logic, circuit, and physical design checks. Release to fabrication. Meanwhile, prepare the final application and system software on the simulated or emulated system model. Perform initial system power-up testing and release the design to final system testing and production.

3.6 Addressing the Six Problems

Experienced SOC designers implement many variations of the above design flow but the development history of many advanced multiple-processor SOCs supports this basic flow. Configurability expands the role of processors in SOC design and enables a new SOC design flow. The

advanced SOC design method advocated here appears to deliver on the promise of significantly easing the six key problems of existing SOC design methods, as introduced in Chapter 2:

1. Changing market needs
2. Inadequate product volume and longevity to support a single-use SOC design
3. Inflexibility in the semiconductor supply chain
4. Inadequate SOC performance, efficiency, and cost
5. Risk, cost, delay, and unpredictibility in design and verification
6. Inadequate coordination between hardware and software design teams

3.6.1 Make the SOC More Programmable

Using this new SOC design method, processors implement a wide range of functions: both traditional software-based functions and traditional hardware-based functions. Extensible processors used as basic building blocks bring greater programmability to all these functions. In traditionally hardwired function blocks, control from high-level-language applications running from RAM displaces finite-state-machine controllers implemented with logic gates. When replacing traditional, processor-based functions, extensibility increases performance headroom and allows incremental features to be incorporated without hitting a performance ceiling.

The processor-centric SOC design approach increases the adaptability of the design at every stage of its lifecycle: during early definition, during detailed implementation, at product launch, and for subsequent generations of end products. Increased SOC adaptability gets the product to market earlier and more profitably and increases the likelihood that the design remains the right product as market requirements evolve.

3.6.2 Build an Optimized Platform 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.

Fig 3-19 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. As Fig 3-19 shows, programmability is useful for all SOCs and is economically essential for lower volume SOCs.

3.6.3 Use Portable IP Foundations for Supply Leverage

The globalization of markets and the emergence of silicon foundries increase the number of silicon supply options available to chip designers. SOC designers working at systems companies have many ASIC suppliers from which to choose. They also have the option of designing the complete SOC themselves using a customer-owned-tooling (COT) VLSI implementation and a silicon foundry. Semiconductor companies are generally moving away from operating their own fabs and are moving to fabless models, to joint venture fabs, or to a mixture of operating models. Flexibility in fabrication choices offers pricing leverage and ensures better fab availability to the chip designer. Making the chip design portable increases the ease of moving between fabs and increases the longevity of the design over process generations.

Portable SOC designs are less silicon-efficient than hand-optimized, fab-specific designs, but automated tools (including both logic compilers and processor generators) generate these designs much more cheaply and quickly. A faster, easier design process lets the designer tune the chip more thoroughly and allows a much wider range of applications to be powered by optimized chips. Application-directed architectures, even in portable design form, are generally much more efficient than more generic designs that rely solely on circuit optimization. Use of extensible processors as basic building blocks complements the leverage associated with process portability. Each programmable block is generally reusable at the design level and aids the generality and reusability of the design as a whole.

3.6.4 Optimize Processors for Performance and Efficiency

The impact of using application-specific processors on data-intensive tasks is usually significant and sometimes dramatic. Five-fold to ten-fold improvement over standard RISC architecture performance is routine. Designers can commonly speed up applications with especially high bit-level and operand-level parallelism by 50 or 100 times using the processor-centric SOC design approach. Application-specific processor architectures sharply reduce the required number of processing cycles required for a given level of application throughput.

In most cases, the expanded configuration has somewhat increased power dissipation on a per-cycle basis but the actual, overall power consumption for any given application drops with cycle count. In fact, a lower operating frequency allows the necessary performance to be reached at lower operating voltage too. As described earlier, this frequency reduction leads to better than linear improvement in real power efficiency. This kind of performance headroom and power efficiency, combined with easy programmability, translates directly into improved end-product attributes, including greater functional capabilities, longer battery life, and lower product cost.

3.6.5 Replace Hard-wired Design with Tuned Processors

No modern digital design is bug-free—large systems are simply too complex to avoid all flaws. The designer's real goal is to reduce the number and severity of bugs and increase the speed and flexibility of fixes. The more a system's functions can be put in soft form—dynamically loadable during operations or upgradeable at system power-on—the more likely it is that bugs can be

fixed without a silicon respin. Just as FPGAs offer fast, cheap upgrades and patches to basic logic functions, the RAM-based code in processor-centric SOCs enables rapid fixes for these more complex, high-volume electronics platforms. In addition, processor-based design offers faster and more complete modeling of the system functions, reducing the number of potential bugs that need fixing. The net result is a higher rate of first-time-functional chip designs and greater longevity for those designs in the marketplace.

3.6.6 Unify Hardware and Software with Processor-Centric SOC Methodology

The hardware and software views of a complex SOC commonly diverge. Three aspects of the new SOC approach mitigate this tendency. First, the development tools allow rapid creation of an early, inexact but functional model of key system behavior. This reference model removes much ambiguity from the early specification and allows gradual refinement of both the software-based functions and the underlying multiple-processor hardware platform.

Second, the description of a system as a set of C/C++ tasks, a set of processor configurations, and the interconnections among the processors constitutes a more abstract, portable, and comprehensible description than the more traditional ad hoc collection of programs, Verilog–coded logic functions, and other block-boxes. Hardware and software developers and system architects can understand and share this system representation and manipulate the representation using a common set of tools that offer both hardware- and software-oriented views.

Third, the integration of subsystems—the Achilles heel of many complex projects—is naturally organized around a processor-based perspective. Integration does not require welding together two innately dissimilar forms of matter (code and gates) but rather involves composition of a set of components of a closely related nature (configured processors with associated task software). The result is smoother integration with fewer communications troubles and fewer surprises late in the design cycle.

3.6.7 Complex SOC and the Six Problems

This new SOC methodology is not a panacea, but it is an effective tool. A more detailed discussion of processor-centric SOC design methods appears in Chapters 4, 5, 6, and 7. Chapter 8 looks to the long-term implications of this new system-design style, projecting deep changes in both the evolution of SOC design and the structure of the industry that creates new chip designs.

3.7 Further Reading

- A quick introduction to Tensilica's Xtensa architecture and basic mechanisms for extensibility can be found in the following: Ricardo Gonzalez, "Configurable and Extensible Processors Change System Design." In *Hot Chips 11*, 1999. The Tensilica Web site (*http://www.tensilica.com*) also offers a wealth of information on the technology.

The following articles are particularly relevant to the history of processor description and tool automation:

- G. Bell and A Newell. “The PMS and ISP descriptive systems for computer structures.” In *Proceedings of the Spring Joint Computer Conference*. AFIPS Press, 1970.
- The GNU C Compiler emerged in the 1980s as the most widely visible effort to create an architecture-neutral compiler technology infrastructure. Its goal was simplified, and it was not designed for fully automated support of new processor architectures. See *http://gcc.gnu.org*.
- J. Van Praet, G. Goosens, D. Lanner, and H. De Man, “Instruction set definition and instruction selection for ASIP.” In *High Level Synthesis Symp*. pp. 1116, 1994.
- G. Zimmermann, “The Mimola design system—A computer-aided digital processor design method.” In *Proceedings of the 16th Design Automation Conference*, 1979.
- H. Corporaal and R. Lamberts, “TTA processor synthesis.” In First Annual Conference of ASCI, May 1995.
- S. Pees, A. Hoffmann, V. Zivojnovic, and Heinrich Meyr. “LISA—Machine description language for cycle-accurate models of programmable DSP architectures.” In *Proceedings of the Design Automation Conference*, 1999.
- Marnix Arnold and Henk Corporaal. “Designing domain specific processors.” In *Proceedings of the 9th International Workshop on Hardware/Software Codesign*, pp. 61–66, Copenhagen, April 2001.
- John R. Hauser and John Wawrzynek. “Garp: A MIPS processor with a reconfigurable coprocessor.” In *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 12–21, 1997.
- Alex Ye, Nagaraj Shenoy, and Prithviraj Banerjee. “A C compiler for a processor with a reconfigurable functional unit.” In *Proceedings of the 8th ACM International Symposium on Field-Programmable Gate Arrays*, pp 95–100, February 2000.

These two articles concentrate on instruction-set description for architectural evaluation rather than for actual hardware generation:

- George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. “ISDL: An instruction set description language for retargetability.” In *Proceedings of the Design Automation Conference*, 1997.
- Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. “LISA machine description language for cycle-accurate models of programmable DSP architectures.” In *Proceedings of the Design Automation Conference,* 1999.

Some degree of configurability has become popular in commercial processor and DSP cores. The following vendor Web sites describe configurable features:

- ARC International’s ARCTangent processors and ARChitect tools are described on the company’s Web site: *http://www.arccores.com*.

- MIPS Technologies' features for user-modification of processor RTL are described on the company's Web site: *http://www.mips.com*.
- CEVA's CEVA-X DSP architecture offers the option of user-defined extensions. See *http://www.ceva-dsp.com/*

HP Labs researchers have written a number of papers on processor generation, including the following:

- Shail Aditya, B. Ramakrishna Rau, and Vinod Kathail. "Automatic architectural synthesis of VLIW and EPIC processors." In *International Symposium on System Synthesis*, pp 107–113, 1999.
- J. A. Fisher, P. Faraboschi, and G. Desoli. "Custom-fit processors: letting applications define architectures." in *29th Annual IEEE/ACM Symposium on Microarchitecture (MICRO-29)*, pp 324–335. 1996.

The exploration of application-specific processors has also included notions of field-configurable processor hardware:

- Rahul Razdan and Michael D. Smith. "A high-performance microarchitecture with hardware-programmable functional units." In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, 1994.
- Michael J. Wirthlin and Brad L. Hutchings. "DISC: The dynamic instruction set computer." In *Proc. SPIE*, 1995.

A more detailed analysis of the impact of parallelism on power efficiency can be found in A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power CMOS digital design." In *IEEE Journal of Solid-State Circuits*, 27(4) April 1992, pp 473–484.

Application benchmarking stimulates rich debate.

- Historically, the most widely cited (and most commonly misused) benchmark used on embedded processors is Reinhold Weicker's Dhrystone benchmark. R. P. Weicker, Dhrystone: A synthetic systems programming benchmark. Communications of the ACM, 27:1013–1030, 1984.
- The EEMBC benchmark suite has become the most widely used modern benchmark for embedded processors. The benchmarks span consumer (mostly image processing), telecom (mostly signal processing), networking (mostly packet processing), automotive, office automation (text and image rendering), Java and 8-/16-bit microcontroller applications. See http://www.eembc.org for a more comprehensive discussion and results.