

**PARALLEL PROCESSING CAN IMPROVE PERFORMANCE, BUT IT CAN ALSO SIMPLIFY YOUR DESIGN BY BETTER REFLECTING THE NATURAL PARTITIONING OF YOUR APPLICATION BEHAVIOR AND DEVELOPMENT RESOURCES.**

# Exploring the anatomy of MULTIPROCESSOR DESIGNS

At a glance .....50  
Exploiting parallelism .....52  
Multiprocessor programming languages .....54  
For more information .....56

**Y**OU CAN DEFINE MULTIPROCESSOR DESIGNS as systems that perform functions and tasks among multiple processors that coordinate and communicate with each other to deliver a coherent behavior. A multiprocessor application is more complex than a

single-processor design. It requires additional programming for housekeeping and coordinating functions, and it is more complex to debug, because of processor interactions that are absent in single-processor architectures. Despite the added complexity, multiprocessor designs have existed for many years as high-performance computers and workstations and are appearing in an increasing number of embedded-system applications.

One obvious reason to use multiple processors is that they provide more processing capability than does a single processor. Some performance considerations for choosing a single- or multiple-processor architecture are the amount of real-time algorithm processing, the re-

quired response time for processing external events, the amount of math-intensive processing you need, and the level of concurrency you require. Multiprocessor architectures afford you some load balancing and the opportunity to consider using tailored processors for each system task.

Consider how real-time processing suffers if there is significant uncertainty of the execution time. For example, when a processor flushes pipelines, experiences cache misses, performs context switches, or employs out-of-order and speculative execution to increase efficiency, the system behavior becomes less deterministic. Likewise, response time for processing external events can suffer if the same processor must also run high-priority, continuous, real-time tasks—especially if

Illustration by Daniel Guidera

those tasks employ multi-issue instructions that block interrupt servicing until they complete executing. Implementing two tasks on separate appropriate processors allows you to divide and conquer the performance and determinism challenges for each type of problem. Disk drives are examples of host-interface functions that execute well with a cached processor and the servo-control portion of the system that executes better on a stripped-down processor.

DSP architectures focus on performing continuous iterative math processing, most notably multiply-accumulates, by supporting rapid data movement via tight interactions with the peripherals. To achieve this continuous processing, these processors often employ complex memory structures, such as multiple buses and mixes of memory types, and specialized instruction sets that include fixed-point math and hardware-accelerated application-specific operations. Although these mechanisms work well for their intended purpose, they perform more poorly than the mechanisms that RISC architectures employ for control and response processing.

The trend toward using programmable building blocks for SOC (system-on-chip) designs is contributing to an increase in processor and core choices that explicitly support embedded-multiprocessor designs. They provide scalable, programmable performance beyond what single-processor architectures can deliver for computationally intensive applications, and they better reflect the natural partitioning of many networking, multimedia, and other embedded systems that are inherently multichannel or convergence applications. For these types of systems, using more than one processor can best balance and meet the project's performance, cost, power-dissipation, risk, and time-to-market goals.

Many embedded designs are employing multiprocessor architectures for reasons other than just pure performance. Some considerations are your legacy software resources and development tools; the stability of key components, such as those supporting evolving standards; simplifying your engineering effort; system security; fault-tolerance requirements; and meeting price, power, thermal, and EMI constraints. Multiprocessor architectures afford you the flexibility and opportunity to apply your most

#### AT A GLANCE

- ▶ Multiprocessor designs are a strategy for scalable, programmable performance.
- ▶ Partitioning your design over several processors can simplify your development effort.
- ▶ Heterogeneous systems can use "best-in-class" processors.
- ▶ Interconnect mechanisms are application-specific and represent a significant part of a design's value.
- ▶ Being able to capture system-level processor interactions is key to validating your system.

appropriate engineering resources to each design task.

Your engineering experiences with processing architectures and the applicability of legacy resources to your current project influences your choice to use a single- or a multiple-processor design (Reference 1). Your legacy code or operating system may dictate your choice of processors, and the way you partition your engineering staff. Using different teams to work on front end, back end, control, and signal processing, for example, affect your processor-architecture choice. Cell phones are an application in which a signal-processing team develops the radio-control and voice-codec functions that execute on a DSP, and an application team develops the human-interface functions to run on a microcontroller. The stability of the standards or protocols, such as for multimedia or wireless applications, influences whether you implement them as software to maintain programmability and flexibility or use some hardware elements to gain power and cost benefits.

#### THE MORE, THE MERRIER

Because software complexity usually scales worse than linearly with code size, designing your software to operate across several dedicated processors can reduce the development time over monolithic implementations, simplify the debugging effort, and improve system reliability. Using separate processors can make the system easier for you to understand the interactions between partitioned tasks and

minimize intertask dependencies from interrupt latency, processor loading, and memory usage. The interaction focus changes from shared intertask resources to shared and dedicated interprocessor resources. You can simplify performance allocations by allowing full operating-system support for portions of the system and little or no operating-system support for others.

Even when a single processor can meet your performance requirements, it may not meet your requirements for cost, power, EMI, and thermal budgets, for example. Using multiple processors, you can drop your clock-frequency ratings to avoid RF frequencies and reduce your design's EMI and thermal profile. You can stretch your system's power budget if the task partitioning allows you to allocate high-speed, burst-oriented processing to a processor and to disable it during periods of inactivity while another processor handles the continuous operations. You can improve system security by moving the security or encryption implementation to a processor that is separate from the main processor and might be running a standard operating system that could simplify reverse-engineering.

Although the benefits of integrating multiple cores in a single device are many, you might choose to physically distribute a multiprocessor architecture across several discrete devices. For example, if your system requires redundancy or fault tolerance, you want to isolate processor failures from one another. Employing a network model of processors is another example of distributing a multiprocessor architecture. This method is appropriate when you need sparse interprocessor communication, minimize signal noise, and reduce system-wiring weight by locating each processor near its relevant processing and signals, such as in an automobile.

It is important to recognize that some applications do not partition easily across multiple processors. If you require tightly coupled communication between partitioned tasks and you cannot envision a suitable interconnection to support the system throughput or latency requirements, you need to reconsider your partitioning or even abandon the multiprocessor approach.

Successfully employing multiprocessor architectures relies on the amount

*(continued on pg 54)*

## EXPLOITING PARALLELISM

The characteristics of your data and control flow significantly dictate how you can exploit parallelism in your system (Reference A). Some single-processor architectures employ pipelining and multiple-instruction-issue techniques to nearly transparently exploit parallelism in the instruction code and increase the amount of work that the processor can perform in a given time. Some processors raise the parallel-abstraction level and support the use of multithreading or SIMD (single-instruction, multiple-data) techniques to increase efficiency (Figure A).

Pipelining is a parallelism technique to exploit temporal instruction-level parallelism and to overlap the execution of multiple instructions by distributing instruction execution over more clock cycles and reducing the amount of work for each instruction each cycle. This technique increases the number of instructions executed in a given time by minimizing the time an execution unit spends idle. Scaling pipelines, by making them deeper, allows you to increase the processor operating frequencies and the amount of work performed during each clock cycle. Unfortunately, deeper pipelines incur longer restart penalties when a pipeline flush occurs, which also decreases the determinism of the system. Another challenge is that long latency operations, such as off-chip memory accesses, can stall the pipeline and lead to more frequent idle execution units and a lower average rate of instruction execution.

Superscalar techniques, or processors that issue multiple instructions per clock cycle, exploit spatial instruction-level parallelism by determining whether an instruction can execute independently of other

sequential instructions. If one instruction depends on another, the instructions must execute in the proper sequence. The processor then uses multiple execution units to simultaneously carry out two or more independent instructions. This technique experiences some of the same efficiency losses as pipelining and experiences a rapid drop-off in benefits when you scale it beyond a dual-issue design; dependency analysis to perform instruction-level parallelism fails to keep the extra execution engines busy for most applications. VLIW (very-long-instruction-word) processors are one way to transfer the task of instruction scheduling from the processor to the compiler to reduce hardware complexity and to improve the diminished returns of scaling beyond a few execution engines. In general, compilers do not have access to runtime information to maintain the processor resources at the highest efficiency.

A thread is an abstraction that allows program code and data for one context to remain independent from other contexts. The system maintains the processing state for each thread

separately from the other threads. Multitasking is similar in concept to multithreading but relies on an operating system rather than the hardware for scheduling. Hardware-assisted multithreading relies on duplicated processor resources to hold the system state for each thread. It can execute rapid context switches, allowing the system to exploit thread-level parallelism by switching to a different thread when other threads stall. The industry has implemented many forms of multithreading, but the goal of each form is to minimize the thread-switching overhead to maximize the pipeline-stall scenarios that multithreading can cover. To benefit from multithreading, the programmer must explicitly identify the thread-level partitioning in the software, because software-development tools and compilers currently do not automate the partitioning of program code into independent threads. Intel's hyperthreading technology is an example of current efforts to extend hardware support for multithreading systems and to provide a virtual processor to multiprocessor-aware operating systems.

Embedded-signal and multi-

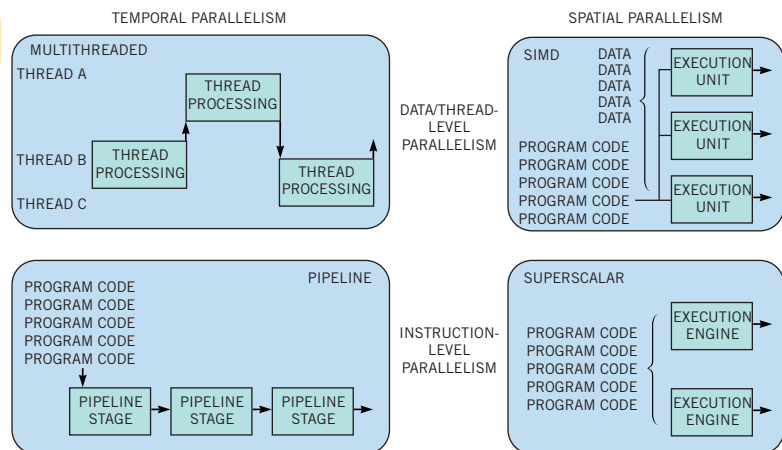
media processing often consists of iteratively applying computationally intensive algorithms on parallel streams of data in a real-time context. SIMD architectures exploit this data-level parallelism to simultaneously apply the same operation, such as a multiply-add, to multiple data streams and to gain significant speed for many critical inner-loop operations. As in multithreading, the programmer or compiler must explicitly identify data-level parallelism for the hardware.

Scaling these techniques to even higher levels of parallelism abstractions forms the basis for structuring multiprocessor architectures. Properly deployed multiprocessor designs can exploit parallelism at the process-level, in which pipelining, multi-instruction-issue, and SIMD techniques are weaker. Key considerations for successful multiprocessor architectures are whether they can continue to scale, maintain programmability, and increase reusability.

### REFERENCE

A. Diefendorff, Keith, and Yannick Duquesne, "New Degrees of Parallelism in Complex SOCs," MIPS Technologies, July 2002.

**Figure A**



**Multiprocessing relies on parallelism in a system. Multiprocessing implementations exploit parallelism by identifying abstraction levels and applying resources along the temporal or spatial dimensions.**

and types of parallelism inherent in your application (see sidebar “Exploiting parallelism”). Many single-processor architectures employ techniques such as pipelining and superscalar instruction-issue, which rely on temporal or spatial parallelism at the instruction-level to increase performance. SIMD (single-instruction, multiple-data) and multithreading techniques allow systems to exploit parallelism at the data- and thread-level of abstraction. Software partitioning is key to successful multiprocessor systems, and such an architecture provides the most gain when you structure it to take advantage of a system’s process-level parallelism.

SMP (symmetric multiprocessing) is a homogeneous processor topology in which the system dynamically distributes the processing workload across a tightly coupled, “share-everything” network of processors. The operating system in such systems is generally aware of each processor and manages the task distribution.

These systems maintain cache-coherent subsystems and can migrate processes, including the operating-system kernel, from processor to processor for load balancing. Scaling the number of processors is limited, because as the number of processors increases, the communication overhead becomes a larger proportion of the total system workload. Network and transaction servers that manage multi-user environments are common targets for SMP architectures.

Another homogeneous processor approach uses multiple, identical processors nonsymmetrically. This method assigns to each processor dedicated, specific tasks to increase the performance or bandwidth of the system. The systems must uniquely identify each processor and maintain a global synchronization mechanism. The homogenous approach is suitable when each processor is executing comparable tasks, or when it is important to stay with a single development environment; otherwise, a heterogeneous ap-

proach may be more appropriate.

Many embedded applications benefit from heterogeneous multiprocessor topologies for systems that apply different operations to different data. Heterogeneous systems can take advantage of “best-in-class” processors to partition and perform the tasks to which each is best suited. A common pairing is to combine a general-purpose processor to manage general systems tasks with a specialized processor to accelerate performance-critical functions or perform real-time processing. A general-purpose processor, such as an ARM, a MIPS, or a PowerPC core, can host the main operating system, configure and start the other processors, manage the user and system interfaces, and execute any application software. The number and types of processors in any multiprocessor design is application-specific and varies widely. For example, choosing a specialized processor can include considering a microcontroller for real-time

## MULTIPROCESSOR PROGRAMMING LANGUAGES

Efforts have been ongoing to incorporate multiprocessing concepts into programming languages. Charles Anthony Richard “Tony” Hoare, PhD, Professor Emeritus at The University of Oxford’s Computing Laboratory first described CSP (Communicating Sequential Processes), a formal language to describe parallel systems (**Reference A**). It treats processes as independent, self-contained entities with specific interfaces that they use to interact with the environment. This approach allows that two processes, combined to form a larger system with a defined interface, constitute a larger process. Much work has continued since the original paper was published.

CSP provides the framework for the Occam programming language. Occam is a parallel processing language that originally operated with the Inmos transputer processor. The language describes concurrent processes that communicate via one-way channels. The process

is the basic entity in Occam, and it includes four fundamental types: assignment, input, output, and wait. Constructed processes specify sequential or parallel execution and the input channel associated with each process. A number of Occam compilers are available for download at <http://wotug.uk.ac.uk>, including back-end translators or front-end translators that convert Occam into portable C. The JCSP (Java Communicating Sequential Processes), CTJ (Communicating Threads for Java) and JavaPP (Java Plug and Play) libraries for Java incorporate some of the principles of Occam.

Unlike C/C++, Java has from the outset addressed support for multithreaded programming. The Java specification requires that the standard libraries are thread-safe and reentrant. The language syntax includes special facilities to support synchronization between threads. When multiple threads run on a single processor machine, the operating system

time-slices their execution. When the same multithreaded application runs on a multiprocessor architecture, true parallel execution of threads is possible.

It is unreliable to share variables in C/C++ that are not declared as volatile, because there is no way to determine when values cached by one thread’s context are visible to other thread contexts. Declaring variables as volatile prevents the compiler from allocating them to machine registers and can generate special coherency instructions each time you fetch or store values. In contrast, Java’s language syntax includes mechanisms, such as synchronized statements, to coordinate, synchronize, and share information between multiple threads.

At the beginning of a Java-synchronized statement, the JVM (Java Virtual Machine) refreshes all of the thread’s cached variables from global shared memory, whether in machine registers or in a processor’s private memory cache. At the end of each

synchronized statement, the JVM copies the thread’s cached variables back to globally shared memory. Associated with synchronized statements are built-in monitorlike services that allow threads to wait for the system to satisfy particular conditions and to notify other threads that particular conditions have been satisfied. Because the semantics for multithreading are a standard part of Java, it is easier for developers to write portable multithreaded applications with Java than with C/C++. However, although Java is stronger for implementing portable, multithreaded designs, it is still weaker than C/C++ for developing deterministic, real-time components.

### REFERENCE

A. Hoare, CAR, *Communicating Sequential Processes*, Prentice-Hall International Series in Computing Science, Prentice-Hall International, Englewood Cliffs, NJ, and London, 1985.

control applications, a DSP for wireless-communication applications, or a VLIW (very-long-instruction-word) processor for media-streaming applications.

### INTERCONNECTING

In multiprocessor systems, the processors communicate and coordinate with each other. All multiprocessing systems require some method of sharing data memory among the processors. This method involves defining private and shared resources, the interconnect interfaces, and the protocols to coordinate everything. You must take care when structuring your system not to block or inhibit accesses to these areas and to employ appropriate arbitration or priority schemes that balance throughput, latency, and scalability. Your goal is to get the right data to the right place at the right time with hard real-time predictability.

On one end of the spectrum, the processors all access and share the same physical memory. Shared memory offers low overhead for area, power, and data-transfer time and suits tightly coupled systems. It supports a simpler programming model, because the software on each processor accesses memory in the same way, but its scalability is limited because congestion and contention increases as multiple processors are simul-

taneously accessing memory. On the other end of the spectrum, each processor has its own local memory. One benefit of using local memory is optimized data-access speed. Another is increased system concurrency, because you avoid the contention issues of fully shared memory systems. A weakness is that you need duplicate resources. Also, the programming model to access data is often system-specific and hinders software reuse, because it can require an explicit rewrite to address differences from one system to another. Hybrid approaches balance pure shared and local memories by implementing semiprivate memories that support an access mechanism for other processors.

Many interconnect-interface implementations accomplish processor intercommunication, but each addresses different system concerns and makes trade-offs at different levels of cost, flexibility, performance, and robustness. Shared-memory systems can be easy to implement. A serial link provides one of the cheapest ways to link processors when performance is not at a premium. A dedicated parallel bus provides the highest performance. You can also use host ports; SPI-link ports; CANs (controller-area networks); and Ethernet, ring, crossbar, and mesh networks to implement inter-

processor communications. Using standardized interfaces, such as interconnect buses, can simplify integrating processors and third-party IP (intellectual property) into your system and reduce your development effort. There is no universally accepted, standard interconnect bus for embedded applications, but a number of specifications, such as HyperTransport, CoreConnect, AMBA, and SuperHyway, are available to support multiprocessor designs.

Interconnect buses are easy to implement, because processors share one bus; however, the available bandwidth per processor decreases as the number of processors on the bus increases. One strategy for increasing the number of processors or the amount of concurrent activity the bus can support, such as with a MIPS high-performance bus, is to use a single interconnect bus that can issue and sequence multiple requests while resolving out-of-order responses. Using a more advanced bus also avoids many of the issues associated with using multiple buses, but the interconnect complexity greatly increases as you add support for more concurrency, such as handling splits, multiple outstanding requests, and out-of-order requests. If your system does not benefit from these capabilities, this type of interconnect can complicate your sys-

## FOR MORE INFORMATION...

For more information on products such as those discussed in this article, go to [www.edn.com/info](http://www.edn.com/info) and enter the reader service number. When you contact any of the following manufacturers directly, please let them know you read about their products in *EDN*.

### Adelante Technologies

+31-40-2353-100  
[www.adelantetech.com](http://www.adelantetech.com)  
Enter No. 364

### ARC

1-408-437-3400  
[www.arc.com](http://www.arc.com)  
Enter No. 365

### ARM

1-408-579-2200  
[www.arm.com](http://www.arm.com)  
Enter No. 366

### Atmel

1-408-441-0311  
[www.atmel.com](http://www.atmel.com)  
Enter No. 367

### Broadcom

1-949-450-8700  
[www.broadcom.com](http://www.broadcom.com)  
Enter No. 368

### First Silicon Solutions

1-503-292-6730  
[www.fs2.com](http://www.fs2.com)  
Enter No. 369

### IDT

1-408-727-6116  
[www.idt.com](http://www.idt.com)  
Enter No. 370

### Infineon Technologies

+49-89-234-00  
[www.infineon.com](http://www.infineon.com)  
Enter No. 371

### Intel

1-800-628-8686  
[www.intel.com](http://www.intel.com)  
Enter No. 372

### LSI Logic

1-866-574-5741  
[www.lsillogic.com](http://www.lsillogic.com)  
Enter No. 373

### Mentor Graphics

1-800-547-3000  
[www.mentor.com](http://www.mentor.com)  
Enter No. 374

### Metrowerks

1-800-377-5416  
[www.metrowerks.com](http://www.metrowerks.com)  
Enter No. 375

### Microchip

1-480-792-7200  
[www.microchip.com](http://www.microchip.com)  
Enter No. 376

### MIPS Technologies

1-650-567-5000  
[www.mips.com](http://www.mips.com)  
Enter No. 377

### Motorola

1-512-895-2000  
[www.motorola.com](http://www.motorola.com)  
Enter No. 378

### Nazomi

**Communications**  
1-408-654-8988  
[www.nazomi.com](http://www.nazomi.com)  
Enter No. 379

### NetSilicon

1-630-577-1590  
[www.netsilicon.com](http://www.netsilicon.com)  
Enter No. 380

### NewMonics

1-630-577-1590  
[www.newmonics.com](http://www.newmonics.com)  
Enter No. 381

### Quadros

1-866-879-7823  
[www.quadros.com](http://www.quadros.com)  
Enter No. 382

### SandCraft

1-408-490-3200  
[www.sandcraft.com](http://www.sandcraft.com)  
Enter No. 383

### STMicroelectronics

1-718-861-2650  
[www.st.com](http://www.st.com)  
Enter No. 384

### SuperH

1-800-691-7374  
[www.superh.com](http://www.superh.com)  
Enter No. 385

### Tensilica

1-408-986-8000  
[www.tensilica.com](http://www.tensilica.com)  
Enter No. 386

### Texas Instruments

1-800-336-5236  
[www.ti.com](http://www.ti.com)  
Enter No. 387

### Toshiba

1-949-455-2000  
[www.toshiba.com](http://www.toshiba.com)  
Enter No. 388

### Wasabi Systems

1-646-638-2424  
[www.wasabisystems.com](http://www.wasabisystems.com)  
Enter No. 389

### Xilinx

1-408-559-7778  
[www.xilinx.com](http://www.xilinx.com)  
Enter No. 390

### SUPER INFO NUMBER

For more information on the products available from all of the vendors listed in this box, enter no. 391 at [www.edn.com/info](http://www.edn.com/info).

tem-verification effort and unnecessarily extend your development effort.

Another strategy for increasing the number of processors or the amount of concurrent activity in a bus environment, such as the AMBA specification defines, is to use multiple or layered buses that rely on simpler interconnect protocols. Using segmented or layered buses allows you to separate bus masters and increase your system's concurrent bandwidth by adding buses as needed. Unfortunately, using multiple buses increases the physical wire-routing overhead, especially as the buses get wider—a larger concern when you are optimizing the total design area. Increasing the number of buses in your system also increases the opportunities for crosstalk issues and may make your system more costly than necessary, because you may be underusing some of the bus bandwidth.

Another key element of interconnecting your processors is communicating data coherency, resource exclusivity, and event notification. The code executing on each processor may share resources, process the results from another processor, pass data to another processor, or demand data from another processor. The basic mechanisms for communication are shared memory, hardware flags, and interrupts. It is critical for you to consider the robustness of the communication protocol you use, so you can avoid data corruption, unnecessary resource blocking, and deadlock conditions. The two basic programming models for multiprocessor communication are shared memory and message passing.

You can most simply describe shared-memory programming as a set of shared-memory regions to which processors read and write data to communicate with other processors. The strength of shared-memory programming is its efficiency. All of the processors can simultaneously gain access to the shared memory if the memory locations they are trying to read from or write to are different. However, data-locking and synchronization mechanisms, such as semaphores, are critical to avoiding race conditions when two or more processors need simultaneous access to the same memory location. Implementing the locking and synchronization mechanisms involves managing exclusive and concurrent read-and-write operations that require careful programming.

Shared-memory schemes can reside at

any level of the memory hierarchy, depending on system requirements, allowing you to make the trade-off between cost, latency, and simplicity. The memory may be physically shared, or an extra hardware layer, the operating system, or software libraries may abstract it to appear shared and manage the data locking and synchronizing. The system may experience extra latency over a system that employs hardware primitives if the software performs all of the communication and coordination.

Message passing requires the software to explicitly send and receive data between the processors. The message-passing programming model generally allows

## **INCREASING THE CHALLENGE OF WRITING FOR MULTIPROCESSOR DESIGNS, AVAILABLE SOFTWARE-DEVELOPMENT TOOLS AND PROGRAMMING LANGUAGES GENERALLY RETAIN A STRONG SINGLE-PROCESSOR BIAS.**

coarser granularity and greater control of the distribution of data among processors than does the shared-memory model. There is potentially less system overhead for an optimized message-passing implementation compared with a shared-memory one, because its design can ensure data exclusivity or coherency. You can implement message passing relatively efficiently on shared-memory systems by using a message-passing library that supports and can operate with both shared- and distributed-memory systems. The main disadvantage of using message passing when parallelizing software is that the programmer must usually make extensive modifications to create an efficient implementation.

The interconnect mechanisms you should use depend on the problems you are solving and where the inherent bottlenecks between the processors will manifest. For tightly coupled systems in which the processors are working on the same data, the processors may need to share the same data memories through a high-

speed bus. For high-performance systems, the processors may benefit from using local data memories and high-speed interconnects for message passing. For loosely coupled systems in which the processors are mostly running independently, there is no need for heavy interaction, so the processors may benefit from private data memory and using a lower cost serial-communication interface.

### **SOFTWARE DEVELOPMENT**

Developing software for single-processor systems is challenging despite the availability of mature development tools and programming languages optimized for such systems. Engineers that are developing embedded software for multiprocessor systems must account for processor interactions that do not exist in single-processor systems. The boot-up and power-control sequences are more complicated. Increasing the challenge of writing for multiprocessor designs, available software-development tools and programming languages generally retain a strong single-processor bias (see sidebar "Multiprocessor programming languages"). Software-development tools are in the early stages of supporting multiprocessor development; the most notable effort is concentrating on the debuggers.

Most multiprocessor-software development is essentially partitioning the problem into separate applications and debugging them as separate debugging sessions—in some cases, with two or more host systems. This scenario includes homogeneous systems, even though the tools are the same for each processor and developed software. Many multiprocessor-aware debuggers offer multiplexed-access to multiple processors through a single JTAG port. They provide a common interface for each processor, but they may use software synchronization between the processors, and you can work with only one processor at a time.

The challenges are even greater for heterogeneous systems because they usually involve multiple vendors with disparate and incompatible tool sets. You must learn each tool, and it is nearly impossible to coordinate them unless the vendors have cooperated and designed them to coordinate. The latencies from using macros and other scripts to try to coordinate the tools are unacceptable. Texas Instruments and Metrowerks have invested significantly in their tool sets to

support multiprocessor development that includes a limited range of processor pairings.

One of the biggest multiprocessor-debugging challenges is breakpoint management. If you are using separate debuggers, will they know how to synchronize the starting and stopping of each processor? Should the system stop all of the processors when it recognizes a breakpoint? What happens to those processors and processes that should keep operating? How quickly should each processor halt after recognizing the breakpoint condition? How does the latency vary among processors for each breakpoint? A common problem to debug is the race condition, at which processors are competing for a shared resource and obtaining bad results because of unsynchronized, simultaneous access. Cross-processor debugging latencies that affect the system's temporal behavior make such race conditions harder to reproduce.

The critical development capability is being able to coherently view and debug the processor interactions. Because multiprocessor systems rely on interprocessor communication, there is an emphasis on defining the communication schemes and ensuring that they are working as planned. System simulation is currently an important tool for gaining greater visibility into multiprocessor systems and verifying their operation. Simulators are especially valuable when interprocessor latencies that halt and correlate the entire system are too large and uncertain to support fine-grained debugging.

#### LOOKING FORWARD

On-chip-debugging support logic is mandatory for complex processor architectures. Incorporating compatible debugging blocks in multiprocessor environments is essential to leveraging tool-vendor-support efforts, especially for heterogeneous designs. The Nexus 5001 Forum ([www.nexus5001.org](http://www.nexus5001.org)) is trying to define and establish a standard interface for on-chip-debugging support.

Debuggers will need to be able to stop only certain processes on one or more processors when the debugging system recognizes a breakpoint condition. Certain processes, such as for motor control, cannot stop; they need to continue operating. On-chip hardware assistance for multiprocessor debugging is beginning

to include support for a breakpoint input from each processor and a separate signal to each processor that will signal a halt to its execution. This type of hardware allows each processor's breakpoint signal to individually affect the other processors and can improve the current latencies for trace synchronization and breakpoint management.

Multiprocessor RTOS support and its integration into RTOS-aware, multiprocessor-aware debugging tools will continue to mature. Supported debugging capabilities will include improved visibility into cross-processor RTOS objects, including global and device objects. Multiprocessor RTOSs may even include utilities to configure customized interprocessor-communication-management engines, because managing such communication is unique to each design implementation.

SOC techniques are contributing to an explosion of multiprocessor designs and enabling a wide exploration of ways to use these designs. Choosing an optimized system-partitioning, interconnect-topology, and interprocessor-communication mechanism for your application is as critical to your design's success as it is unique. As a result, system-level industry standards for implementing embedded-multiprocessor designs are neither obvious now nor in the near future. In spite of no standard approach for these designs, people are already extending process-level parallelism abstraction to explore clusters of multiprocessor systems for even more complex high-performance embedded systems. □

---

#### REFERENCE

1. Cravotta, Robert, "Control and signal processing: Can one processor do it all?" *EDN*, March 7, 2002, pg 63.

---

#### AUTHOR'S BIOGRAPHY



*Technical Editor Robert Cravotta has worked on various multiprocessor designs, such as one that combined four DSPs to implement a real-time vision-processing subsystem. You can meet him on Nov 18 to 21 at Embedded Systems Conference in Boston, or you can reach him at 1-661-296-5096, fax 1-661-296-1087, e-mail [rcravotta@edn.com](mailto:rcravotta@edn.com).*