

TEE UP YOUR

MULTIPROCESSING OPTIONS





A GOOD PROCESSOR TAXONOMY HELPS YOU UNDERSTAND PROCESSING ARCHITECTURES' SWEET SPOTS. YOU SHOULD PROBABLY LOOK AT MULTICORE OPTIONS IN THE SAME WAY.

Vendors categorize single-processor architectures by the types of applications they target. For example, microcontrollers, DSPs, network processors, and DSCs (digital-signal controllers) employ different implementations to meet the needs of their target domains. Classifying processors at this top level enables designers to quickly identify candidate processors without delving into the implementation details

of every processor. Differences in implementation details within a processor category become differentiators with a much shorter list of candidates as a result. Categorizing processors in this way allows different versions of each type of processor, such as low-power DSPs and microcontrollers, to use the same implementation techniques without letting that implementation feature confuse what the target is. This issue is one of the points of confusion affecting a contemporary multicore taxonomy.

Another point of confusion is that the term *multicore* has a different meaning depending on who is using it. One use of the term describes multiple processors within a system. Another use constrains the multiple cores to one package, and yet another usage further constrains the multiple cores to one die. People also use the term to refer to identical cores or heterogeneous cores in each of these contexts. These usages fail to converge on one meaning, even though these differences, in many cases, are basically transparent to the

software-development effort. This article uses the loosest interpretation of *multicore* because a chip-level implementation today was a board-level implementation yesterday, and that difference does not change how you build the software. Less transparent to the software-development effort, however, is the application domain; each type of processing, most of which the single-core taxonomy captures, requires different analysis and programming knowledge, and different tools and libraries serve them.

Some multicore processors place their primary focus on performance by stating how many cores they support, how much bandwidth they can handle, and whether they support SMP (symmetrical-multiprocessing) and AMP (asymmetrical-multiprocessing) configurations. As relevant, they also disclose information about coherency mechanisms, shared-data performance, multithreading support, operating-system support, and the ability to balance workloads between cores. You may have to infer the target application and the favored trade-offs from the information the vendor offers.

In contrast, some multicore offerings, similar to single-core offerings, place their primary focus on identifying their target application, which is often digital media, networking, wireless infrastructure, and servers. They then get into the implementation details, such as coherency or processor interconnects, to differentiate them from other similar multicore offerings. However, multiprocessing is not limited to this short list of applications. Embedded-system designers have for decades been quietly and successfully implementing multicore designs. To understand how a more-inclusive multicore taxonomy might look, you should explore a taxonomy for single-core architectures and how you might extend

it to incorporate multicore offerings.

Each of the single-core processor architectures available best fits into a sweet spot of processing complexity (Figure 1). Although each type of processor can perform other types of processing, they are best at the type of processing that the target needs the most. The proposed primary characteristics for mapping each processing sweet spot are computational load and number of states, or contexts. Both of these characteristics are measures of complexity. Computational load can indicate the peak magnitude, total amount, or sustained amount of processing performance the system needs within a system cycle. The number of states can indicate internal system states, number of system inputs and outputs, or a level of possible states or contexts that the system must support. The reason for proposing loose definitions for these characteristics is to rein in complexity and accommodate the range of processing scenarios that include managing scalar data or control flows, managing aggregate data or control flows, and managing many channels of scalar or aggregate data or control flows. Each of these characteristics is implementation-independent and permits an orthogonal differentiation for each type of processor. Each of these processor types evolved over time

AT A GLANCE

- ▣ Processing architectures choose optimization preferences that target a processing sweet spot.
- ▣ Each type of processing has its own development ecosystem.
- ▣ Focusing on functional capabilities enables stakeholders to converge their efforts to evolve their type of processing.
- ▣ Multicore discussions should expand beyond digital media, networking, and server applications.

at different rates, and each trades one or more measures of performance to maximize one or more other measures of performance.

SWEET SPOTS

Microcontrollers are specialized processors that offer a cost and power-efficiency advantage at the expense of flexibility and processing performance. They provide a cost advantage by incorporating memories and peripherals in the same package. They provide a power-consumption advantage partly because they support lower clock rates and partly because they implement only the minimum set of circuitry to perform control processing. If a design needs the flexibility of a larger or a smaller memory, different peripheral set, or higher clock

rate, the design must swap out the processor for another. To accommodate this change, processor vendors offer families of devices with many variations of the same microcontroller that include differing amounts of memory, peripheral sets, and supported clock rates so that developers can realize the best cost and power efficiency without losing flexibility.

The processing sweet spot for microcontrollers is systems that must respond—often deterministically—to external real-world events, such as for motor control, with tight latency requirements. Microcontrollers are capable of rapid, frequent, and prioritized context switching; they usually can also handle many different contexts. A common differentiation for handling context switching includes employing specialized hardware-interrupt processing that can sense and react to external events and change the microcontroller's internal context within a few clock cycles. Alternatively, the system can avoid context-switching overhead by using peripheral-to-peripheral communication coprocessors that are independent of the processor core. Some microcontrollers differentiate for deterministic operation by not using caches or pipelines, so as to avoid the uncertainty from stalls during cache misses or pipeline flushes.

Microcontroller suppliers differentiate their devices with features such as vertically appropriate peripheral sets, on-chip memories, a range of clock rates, packaging options, power-management options, deterministic operation, and development support. Because projects that include microcontrollers are often cost-sensitive, it is important to include only the necessary peripherals and amount of on-chip memory to complete the project. To support higher clock rates, microcontroller suppliers can employ many techniques, such as wide data buses, to mask the access latency to on-chip flash memory. Some of these approaches focus on burst/peak performance versus sustained performance before encountering a wait state. Power management is common in microcontrollers with many ways to implement sleep and low-power modes that have varying granularity for turning off parts of the processor's resources. Each of these differentiations is an example of implementation details that do not

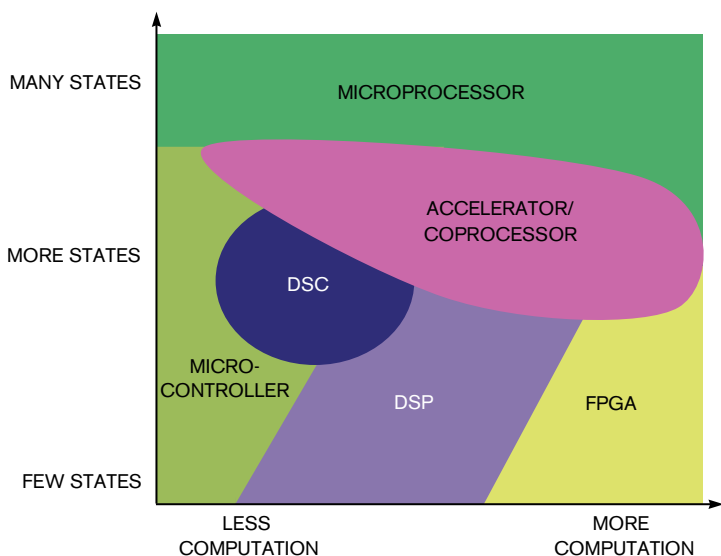


Figure 1 This taxonomy mapping highlights processing sweet spots of mainstream processing architectures and how embedded developers can combine them to complement each other and cost-effectively cover the range of an application's processing requirements.

change the primary function of the microcontroller but enable it to compete with other microcontrollers.

Software development with microcontrollers often involves open- or closed-loop control algorithms as well as filtering of real-world inputs. Microcontrollers are common in board- and system-level multicore embedded designs in which they find use in distributed or periodic system monitoring, distributed control, management of simple user interfaces, and even low-power-supervisory functions to power up and down more expensive parts of the system. They are also present in some multicore chips alongside a DSP as an efficient way to complement the strengths and weaknesses of the DSP architecture.

DSPs are similar to microcontrollers in that they offer advantages in cost and power efficiency at the expense of flexibility and processing performance. However, they substantially differ from microcontrollers because they sacrifice efficiency in handling context switching to maximize their performance of continuous and repetitive calculations, such as in most signal-processing tasks. They also do not integrate many peripherals because they are not ideally structured to handle the context switching that handling many peripherals could require. The most common integrated peripheral is an ADC that the DSP uses to collect a stream of real-world data to process. DSPs often support fixed-point operations to optimize the performance and energy efficiency for fractional mathematics, and they use special address generators to optimize algorithms that work on arrays, circular buffers, or even bit-reversed values. They employ multiple buses and memory structures so that they can do simultaneous memory operations to support continuous single-cycle MAC (multiply/accumulate) operations. They may employ specialized registers to minimize memory accesses and to enable zero-overhead looping.

The processing sweet spot for DSPs is systems that process continuous and sustained streams of data, especially if there is extensive computational processing on that data stream. The upper limit of this sweet spot overlaps with FPGAs, except for highly computationally intensive algorithms that include a high level of state or decision complex-

ity. VOIP (voice over Internet Protocol) is an algorithm that would work better on a DSP than as an FPGA-only implementation because the domain-specific knowledge for compressing data involves many control states and decisions. Software development with DSPs usually requires an understanding of signal-processing algorithms. Many application- and domain-specific analysis and development tools and libraries are available to assist developers with their DSP applications.

DSP suppliers can differentiate their devices by including application-spe-



cific hardware accelerators or by implementing a SIMD (single-instruction/multiple-data) or VLIW (very-long-instruction-word) architecture that supports the simultaneous operation of multiple execution units. Contemporary DSPs use orthogonal instruction sets so that software developers can use compiled C, rather than assembly, to program most of the code. Programming signal-processing code requires different experience and skills from those for developing control or application code. An important differentiator for DSPs is access to signal-processing libraries that either help jump-start a project or enable an application- or control-software developer to use the DSP as a coprocessor without understanding how to program those algorithms.

DSPs commonly find use in embedded-multicore designs; designers often pair them with a complementary microcontroller or microprocessor for control and application-level support. They also often combine them with hardware accelerators or FPGAs to perform heavy lifting for intensive computational pro-

cessing. Another common multicore configuration involves a cluster or array of DSP cores that work together on multiple streams of data or wide streams of data, such as for image processing. The arrays of DSPs can involve dozens of cores that connect at the board level. A lot of experimentation with multicore-DSP configurations has occurred over the years. The commercial failure—versus obsolescence—of many of those attempts serves as a reminder that chip-level multicore offerings have to balance between technological advantage and serving a set of applications that generate enough volume to support the device.

DSCs are new to this taxonomy; most of the market in 2007 formally acknowledged them as separate types of processor. Before then, market participants usually referred to them as “hybrid,” or “unified,” processors because they combined features of DSPs and microcontrollers in a single processor. The processing sweet spot for DSCs focuses on applications that must handle a fair amount of not only context switching, but also signal processing. These devices can alleviate the need to use two heterogeneous cores—a microcontroller and a DSP, for example—in a design. This approach can lower the system BOM (bill-of-materials) cost, and it can lower the development-tools cost because software developers can use the same development tools to program the system. The software developer still must understand the details of programming control or signal-processing algorithms, however.

Because DSCs are relatively new, designers do not have a rich base of experience in implementing them in multicore designs. An early motivation for developing DSCs was to replace multicore designs involving a microcontroller and a DSP with a single device executing a single instruction thread that included both signal-processing and control code. Dual DSCs are now available as single chips. One rationale for using multiple DSCs in a design is that you could normally use one core as a control processor and the other as a signal processor. When one function heavily outweighs the other function, however, the system can more easily perform load balancing between the cores because all of the cores

can support both types of processing.

FPGAs provide a programmable platform that can leverage arbitrarily wide signal-processing algorithms acting as hardware-acceleration blocks. This feature gives them an advantage over DSPs when the signal-processing algorithm is sufficiently wide enough that it can efficiently use more than the available processing units in the DSP. FPGA signal-processing blocks work well when they involve few decision states and large amounts of processing per data point. As the number of decision states increases in an algorithm, it can make sense to place one or more processor cores in or next to the FPGA fabric. FPGA signal-processing implementations function well as coprocessors or accelerators next to a DSP, a microprocessor, or both.

Another processing option is to employ specialized hardware blocks or processors that accelerate software functions. These blocks or processors are usually not stand-alone, instead typically connecting to another processor, such as a DSP or a microprocessor. Some processor vendors integrate these specialized hardware blocks as coprocessors into the same device and even into the instruction-pipeline flow to differentiate their devices. Besides the quick and energy-efficient performance of some tasks, hardware accelerators help offload computational load from the main processor, thus freeing that processing capacity for some other differentiating function. This taxonomy currently lumps network, graphic, and video processors in with the hardware-accelerator and coprocessor category. These types of accelerators are usually not stand-alone systems, and, when they are stand-alone, they exist in a multicore structure, and the proposed multicore categories encompass them. Please contribute to the companion blog post for this article at www.edn.com/blog/1890000189.html to expand on these architectures or add any others.

Microprocessors round out the taxonomy. They employ general-purpose architectures that enable them to perform well enough across the range of processing tasks. They do not handle context switching as quickly or deterministically as microcontrollers do because they usually employ software within the interrupt handling. They do not handle looped

processing as quickly or efficiently as DSPs do because they usually neither include zero-overhead looping nor employ the bus and memory structures to keep a single-cycle MAC busy every cycle. Microprocessors are ideal when the processing requirements that the system must support are not well-known, such as when the system supports user-loaded applications.

Microprocessors generally support large memory-address spaces and rely on large on-chip caches to compensate for time penalties from off-chip memory accesses. They can execute complex operating systems and support a range of legacy code. Microprocessors are also



appropriate for “quick and dirty” prototyping and proof-of-concept exploration when cost and energy efficiency are less important than a short development cycle.

The sweet spot for microprocessors is to support systems that exhibit significant uncertainty in processing behavior and loads, such as when executing disparate processing threads over the same processor resources. A microprocessor’s flexibility and operating-system abstractions of the hardware peripherals make them ideal for the development of sophisticated user interfaces and high-level application code. Microprocessors can perform most functions—at higher cost and energy consumption—that the other specialized processor architectures can perform. Using a microprocessor for control or signal-processing tasks can make sense if the design needs a microprocessor anyway and there is sufficient head room to accommodate the processing; otherwise, for embedded systems, designers may migrate well-defined tasks to the appropriate specialized processor

to save cost and energy consumption.

Multicore-microprocessor designs include desktops and servers. As cell phones evolve to include sophisticated graphics and user interfaces, microprocessors and graphics accelerators are supplanting the microcontrollers that sit next to the DSP that handles the signal processing in these devices. In all of these cases, each processor architecture has use cases for single-core and multicore designs, and, in many cases, the multicore-use cases involve combining processor architectures so that they complement each other.

MULTICORE

This taxonomy highlights the major single-core architectures and the distinct development ecosystems that exist for each type of processing architecture. Each ecosystem includes its own silicon offerings, domain-specific analysis and development tools, and libraries that abstract some of the complexity for those developers who specialize in the other types of processing. Perhaps looking at multicore offerings in a similar fashion will help our industry address a problem that still seems intractable.

A number of companies have come and gone that offered their vision of a multicore architecture. Were they off-base, or were they just missing a single component that could have made all the difference for success?

Multicore terminology is not standard; this article proposes a couple of multicore-configuration names. The terminology instead attempts to propose functional-level descriptions for multicore architectures. A developer should be able to incorporate one or more of these types of multicore architectures in the same design, much as many embedded designs incorporate more than one type of processor architecture.

Channel-multicore architectures target the parallel nature of some applications, such as infrastructure-networking equipment, which apply the same type of processing across many input sources at once. The processing for each input is relatively independent of the processing of the other inputs. These systems implement multiple copies of the processing engine so that each copy can apply the same processing across multiple



channels of data. The system contains duplicate processing resources so that they are local to each processing engine; this approach provides lower latency and better processing performance than do architectures that access the same resources through a shared-access mechanism. The nature of the processing engine is application-specific, and it is not limited to a single function. Software development for these systems focuses on optimizing the domain-specific processing of the inputs so that the system minimizes how much it must manage program resources.

Aggregate-multicore architectures target processing tasks, such as video or image processing, that are impractical or less practical to perform in one instruction thread. The processing engine consists of multiple homogeneous or heterogeneous processing elements that perform separate parts of the same overall task. The architecture aggregates the results from the separate processing elements in some fashion to complete the processing. This type of multicore system has existed for decades as high-end signal processing or simulation systems. These types of systems are of increasing importance for high-performance processing as processor clock rates have stalled and are no longer racing to higher rates. Much research is going into how tools can help developers extract parallelism in their software. No widely used tools are currently available that automate parallelizing general code. Some domain-specific analysis tools for signal-processing-algorithm exploration assist domain experts in determining how to structure their algorithms, but they do not completely automate the parallelization of algorithms and software to take advantage of these types of systems.

Multidomain multicore architectures target those applications that encompass multiple software domains, such as cell phones, automobiles, and many consumer products. This area is the bread and butter of many embedded-multicore designs that developers have been doing for decades. Some coupling and communication may occur between the cores in the system. Each processing element in the system not only makes optimization choices that align with the function it performs, but also works in tandem with and complements the other process-

✚ For related blog posts about embedded processing, go to www.edn.com/blog/1890000189.html.

✚ For a related article about processing options, go to www.edn.com/article/CA6298267.

ing elements in the system. These types of systems have existed as proprietary board-level designs. Single-chip, multi-domain multicore devices are available as vertically targeted devices for applications such as mobile devices with rich video support. Software developers are aware of each type of processing engine they are targeting, and they use the appropriate set of analysis and development tools to perform their tasks. There are challenges in debugging these types of systems because each processor has different on-chip resources and may not provide visibility to all of those resources in the same way from chip to chip.

Feedback-multicore architectures target those applications, such as autonomous systems, that employ feedback within and between subsystems or different domains of the system to perform their functions. These systems may employ a central command unit or aggregate decisions across distributed decision makers. The feedback loop may be as simple as an actuator sense-and-control loop, or it may involve larger-scope feedback that can assist the system with trending, predicting, planning, and learning. These emerging designs currently exist mostly in the academic, aerospace, and military domains, but they are set to spill into the industrial and consumer areas. A candidate industrial system is a smart building that manages systemwide resources across distributed decision makers. Consumer candidates include automated active functions in automobiles as well as household appliances, such as vacuum cleaners. Software developers need an integrated and correlated aggregate of analysis and development tools that cover filters, signal processing, control algorithms, and closed-loop simulations.

CONVERGENCE

This taxonomy for single-core architectures has successfully enabled the industry to converge the vocabulary and key requirements for each application

domain between the relevant chip architects, board-level-system designers, software developers, and development-tool providers. Each processor type has its own ecosystem with its own set of vendors for chip-, board-, and software-level-design support. Even for those hardware and software companies that offer products and services across multiple types of processor, different teams support each processor market.

The multicore-system market could benefit from a convergence of vocabulary and requirements that employs functional capabilities rather than raw implementation and performance details. Rather than a single general-purpose discussion on SMP versus AMP, there should be at least four and at least one for each of the multicore categories. You could say the same for core topologies, coherency mechanisms, messaging, multithreading, operating-system support, virtualization, hypervisors, code parallelizing, and balancing workloads. By defining and establishing the optimization preferences employing required functional capabilities, chip architects, board-level-system designers, and software developers can meaningfully discuss how to allocate technical approaches among all of the stakeholders rather than an ad hoc fitting together of what each group builds. **EDN**

FOR MORE INFORMATION

Altera www.altera.com	MIPS www.mips.com
AMCC www.amcc.com	National Instruments www.ni.com
ARM www.arm.com	QNX www.qnx.com
Atmel www.atmel.com	Radisys www.radisys.com
Freescale www.freescale.com	Tensilica www.tensilica.com
Intel www.intel.com	Texas Instruments www.ti.com
Microchip www.microchip.com	Xilinx www.xilinx.com
Microsoft www.microsoft.com	



You can reach
Technical Editor
Robert Cravotta
at 1-661-296-5096
and rcravotta@edn.com.