A New Epoch for Microprocessors and MPSOC Design

By Steve Leibson and James Kim, Tensilica, Inc.

A New Epoch for Microprocessors and MPSOC Design

By Steve Leibson and James Kim
Tensilica, Inc.

Configurable processors mark the start of a fourth epoch for microprocessors—an epoch that's far more suited to SOC design. Each microprocessor epoch lasted roughly 10 years and produced processors that were suited to their time. In the first epoch, roughly the 1970s, microprocessors simply struggled for competency as they evolved from the early 4-bit processors to the early 16- and 32-bit microprocessors that paved the way for the breakthrough PCs and workstations of the 1980s. PC and workstation growth drove the second microprocessor epoch in the 1980s. The RISC epoch occurred during the 1990s.

During this epoch, even stalwart CISC architectures such as the x86 became RISC architectures in disguise. Each these first three epochs saw growth and development of the microprocessor as a fixed, standalone, reusable block created by microprocessor design specialists. However, the rise of ASIC and SOC manufacturing technologies in the 1990s has laid the groundwork for a fourth epoch in microprocessors—the post-RISC, configurable processor epoch.

Development tools are now advanced enough to allow any designer to tailor a microprocessor core for specific application tasks. Processor tailoring creates processor cores for specialized tasks on SOCs in minutes—a shockingly short amount of time. Because of this rapid ability to tailor processors for specific tasks, configurable processors make excellent, high-performance building blocks for SOC design and can often be used to quickly create functional blocks that might otherwise require months of manual labor to create hand-crafted RTL.

As a result of their high-performance and rapid-developmental abilities, a wide range of end products already incorporate SOCs based on multiple configurable processor cores (multiple-processor SOCs or MPSOCs). These end products range from the biggest and the smallest network routers to consumer electronics products such as camcorders, printers, and low-cost video games.

Two recent developments have embedded configurable processors even further into the mesh of SOC design: fully automated, application-specific instruction-set tailoring and multiple-port access to the processor's internal execution units.

The former development allows SOC designers to more fully focus on system architectural issues while relying on automated tools to labor over the details of achieving performance goals for individual functional blocks. The latter development permanently shatters the formerly ironclad bus bottleneck that has choked microprocessor performance since the first microprocessor appeared in 1971.

Automatic Processor Tailoring

For more than a decade, hardware designers have struggled to transform system specifications written in C or C++ into efficient hardware. The initial system specifications are often written in C and C++ because using these languages produces a spec that can be executed and evaluated on inexpensive PCs.

However, even inexpensive PCs are not suitable many embedded systems designs, especially in the consumer electronics arena, because they cost too much and draw too much power. Consequently, designers have continued their quest to find a tool that converts executable specifications written in C or C++ to hardware.

A variety of approaches—with buzzwords such as “behavioral synthesis,” “C-language hardware synthesis,” and “ESL”—have all fallen short of the mark because they all try to solve what is essentially an intractable problem: transforming a description written in a sequentially executable language into a parallel collection of interoperating, non-programmable hardware blocks.

Tensilica’s XPRES Compiler tackles this design problem using a simpler, more direct approach. Instead of attempting to create application-specific hardware from scratch, the XPRES Compiler starts with a fully functional microprocessor core and then adds hardware to it in the form of additional execution units and corresponding machine instructions to speed processor execution for the target application.

Thus the XPRES Compiler starts with a working hardware design (the Xtensa microprocessor core) and makes it run faster for the targeted application code. The result of this search is a set of microprocessor configurations with a range of performance/cost characteristics presented on a curve as shown in Figure 1.

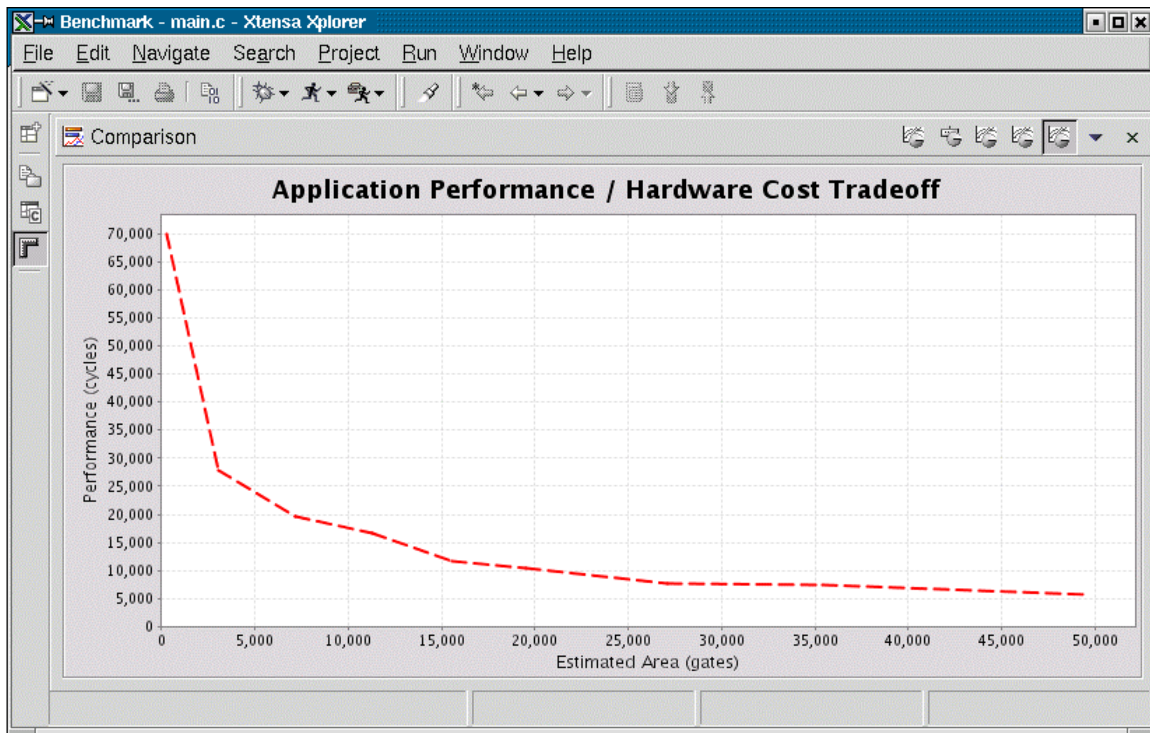

Figure 1: The XPRES Compiler presents the designer with a series of microprocessor configurations that provide increasing amounts of application-specific performance for an increasing amount of silicon area.

Three Paths to Performance Optimization

The XPRES Compiler uses three techniques for creating optimized Xtensa processor configurations: operator fusion, SIMD (vectorization), and FLIX (flexible length instruction extensions). Operator fusion notes the frequent occurrence of sequences of simple operations in program loops.

The XPRES Compiler combines these instruction sequences into one enhanced instruction, which accelerates code execution by reducing the number of instructions executed within the loop. Figure 2 shows an operation dataflow graph that was generated by the XPRES Compiler. It has marked the operations in gray as fusible.

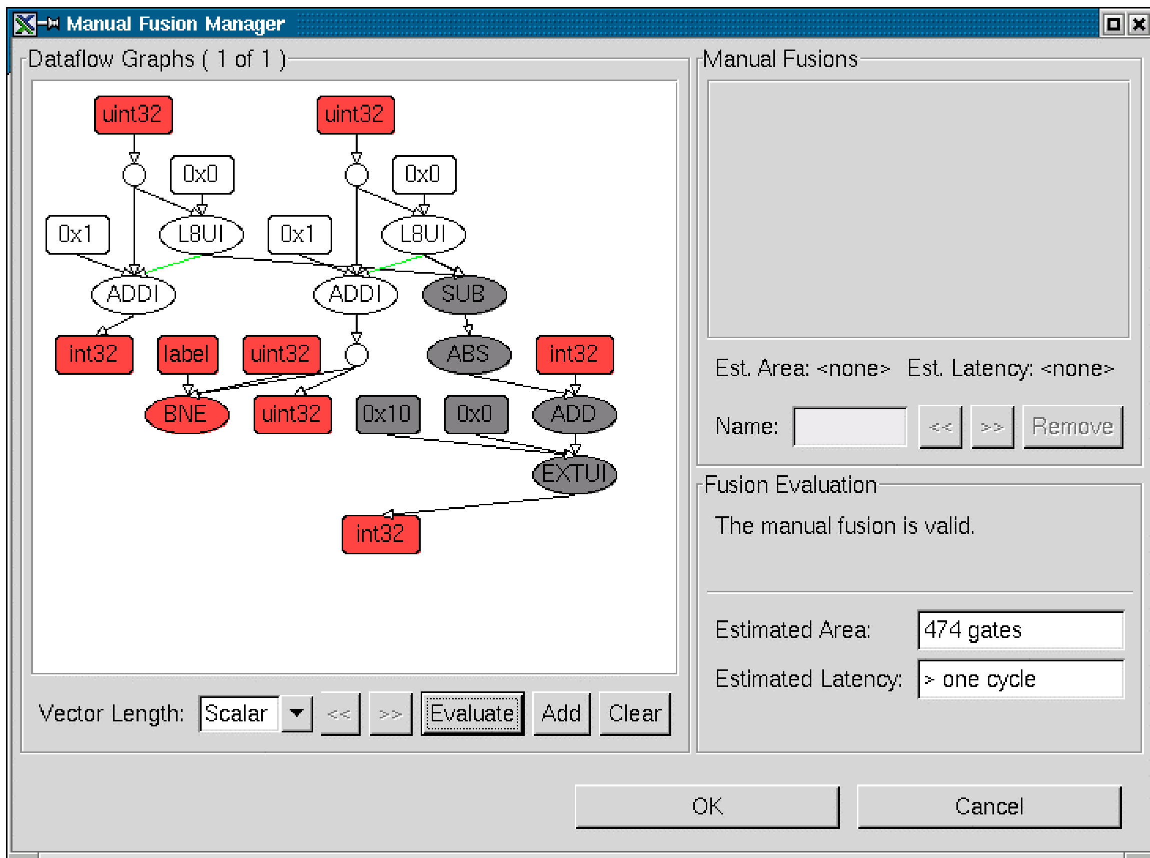

Figure 2: This dataflow graph generated by the XPRES Compiler shows a series of operations marked as fusible. The XPRES Compiler estimates that a new instruction that fuses the subtraction, absolute-value, addition, and bit-field-extraction operations will require 474 additional gates.

Many loops within application programs perform the same operations on an array of data items. The XPRES Compiler can vectorize such loops by creating an instruction with multiple identical execution units that operate on multiple data items in parallel. The addition of SIMD instructions to an Xtensa processor dovetails with Tensilica's XCC C/C++ compiler, which has the ability to unroll and vectorize the inner loops of application programs. The loop acceleration achieved through vectorization is usually on the order of the number of SIMD units within the enhanced instruction.

The third technique used by the XPRES Compiler to accelerate code is Tensilica's FLIX (flexible-length instruction extension) technology. FLIX instructions are multi-operation instructions like fused and SIMD instructions. However, FLIX instructions consist of multiple *independent* operations, in contrast with the dependent multiple operations of fused and SIMD instructions. Each operation in a FLIX instruction is independent of the others and the XCC C/C++ compiler can pack independent operations into a FLIX-format instruction as needed to accelerate code.

Multiple-processor SOC Design with configurable processors

It's a rare application today that can achieve performance goals with just one processor, even with a configurable processor that's tailored for the target applications. However, the MP instruction sets, high-bandwidth interfaces, and small size of configurable processors encourage their use in large numbers in SOC designs.

The choice of hardware-interconnection mechanisms among processor blocks in an SOC greatly affects performance and silicon cost and these hardware-interconnection mechanisms must directly support the interconnection requirements of the MP system design. Message-passing software communications have a natural correspondence to data queues. Similarly, the shared-memory software-communications mode has a natural correspondence to bus-based hardware. Configurable processors offer significant flexibility in supporting arbitrated access to shared devices and memory. The basic topologies for shared memory buses are:

1. **Remote global memory accessed over a general processor bus:** The processor implements a general-purpose interface that allows a wide variety of bus transactions. If the processor determines that that the corresponding data is not local during a read (based on the address or due to a cache miss), the processor must make a non-local reference. The processor requests control of the bus, and when control is granted, sends the target read address over the bus. The appropriate device (for example, memory or input/output interface) decodes that address and supplies the requested data back over the bus to the processor, as shown in Figure 3.

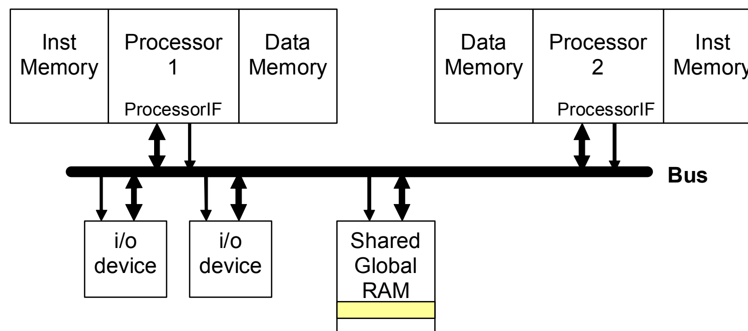

Figure 3: Two processors access shared memory over bus

When two processors are communicating through global shared memory on the bus, one must acquire bus control to write the data; the other processor must later acquire bus control to read it. Each word transferred in this fashion requires two bus transactions. This approach requires modest hardware and maintains high flexibility, because the global memories and input/output interfaces are accessible over a common bus. However, the use of global memory does not scale well with the number of processors and devices, because bus traffic leads to long and unpredictable contention latency.

2. **Local processor memory accessed over a general processor bus:** Configurable processors may allow local data memories to participate in general-purpose bus transactions. These data memories are primarily used by the processor to which they are closely coupled. However, the processor controlling the local data memory can operate as a bus slave, as shown in Figure 4.

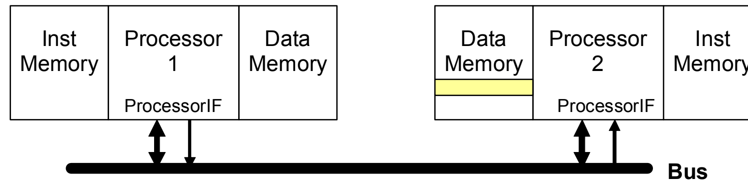

Figure 4: One processor accesses local data memory of a second processor over bus

Latency asymmetry between Processor 1 and Processor 2 encourages *push communication*: when Processor 1 sends data to Processor 2, it writes the data over the bus into Processor 2's local data memory. If the write is buffered, Processor 1 can continue execution without waiting for the write to complete. Thus the long latency of data transfer to Processor 2 is hidden.

3. **Multi-ported local memory accessed over local bus:** When data flows in both directions between processors and latency is critical, a locally shared data memory is often the best choice for inter-task communications. Each processor uses its local data memory interface to access a shared memory, as shown in Figure 5. This memory could have two physical access ports (two memory references satisfied each cycle) or could be controlled by a simple arbiter.

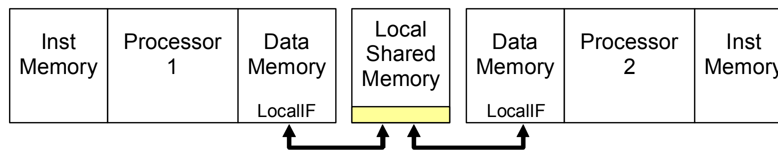

Figure 5: Two processors share access to local data memory

Direct Connect Ports

Direct processor-to-processor connections reduce cost and latency for communication. They allow data to move directly from one processor's registers to the registers and execution units of another processor. A simple example of direct connection is shown in Figure 6. Whenever the Processor 1 writes a value to the output register, usually as part

of some computation, that value automatically appears on the output pins of the processor. That same value is immediately available as input value to operations in Processor 2.

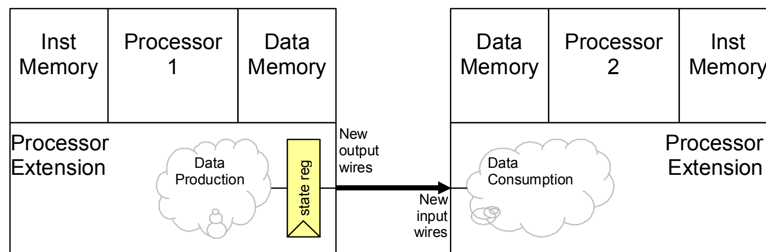

Figure 6: Direct processor-to-processor ports

Data Queues

The highest-bandwidth mechanism for task-to-task communication is hardware implementation of data queues. One data queue can sustain data rates as high as one transfer every cycle or more than 10 Gbytes per second for wide operands (tens of bytes per operand at a clock rate of hundreds of MHz). The handshake between producer and consumer is implicit in the interfaces between the processors and the queue's head and tail.

Application-specific processors allow direct implementation of queues as part of their instruction-set extensions. An instruction can specify a queue as one of the destinations for result values or use an incoming queue value as one source. This form of queue interface, shown in Figure 7, allows a new data value to be created or used each cycle on each queue interface.

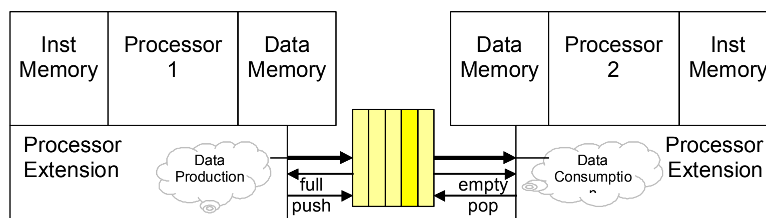

Figure 7: Hardware data queue mechanism

Queues decouple the performance of one task from another. If the rate of data production and data consumption are quite uniform, the queue can be shallow. If either production or consumption rates are highly variable, a deep queue can mask this mismatch.

Because queue interfaces to processor execution units is an unusual feature of commercial microprocessor cores, it's important to discuss this interface mechanism in

more depth. Queue interfaces are added to an Xtensa LX processor through the following TIE syntax:

```
queue <queue-name> <width> in|out
```

The name of the queue, its width, and direction are defined with the above syntax. One Xtensa LX processor can have more than 300 queues, of variable width up to 1024 bits each. Using queues, designers can trade off fast and narrow processor interfaces with slower and wider interfaces to achieve bandwidth and performance goals.

Figure 8 shows how TIE queues are easily connected to simple Designware FIFO's. TIE queue push and pop requests are gated by the FIFO empty and full status signals to comply with the Designware FIFO specification.

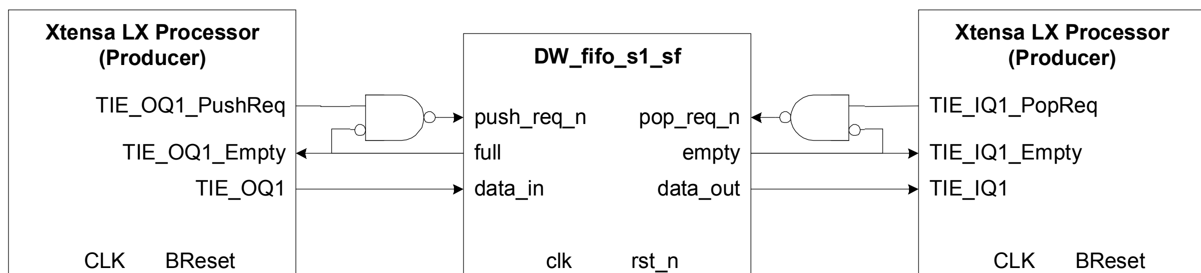

Figure 8: Designware synchronous FIFO used with TIE queues (The diag_n input is driven high, and the almost_full, half_full, almost_empty, and error outputs are unused)

TIE queues serve directly as input and output operands of TIE instructions, just like a register operand, a state, or a memory interface. The following TIE syntax creates a new instruction that accumulates values from an input queue into a register file.

```
operation QACC {inout AR ACC} {in IQ1} {
     assign ACC = ACC + IQ1;
}
```

Figure 9 shows how TIE queues can be used just like other instruction operands in an Xtensa LX processor.

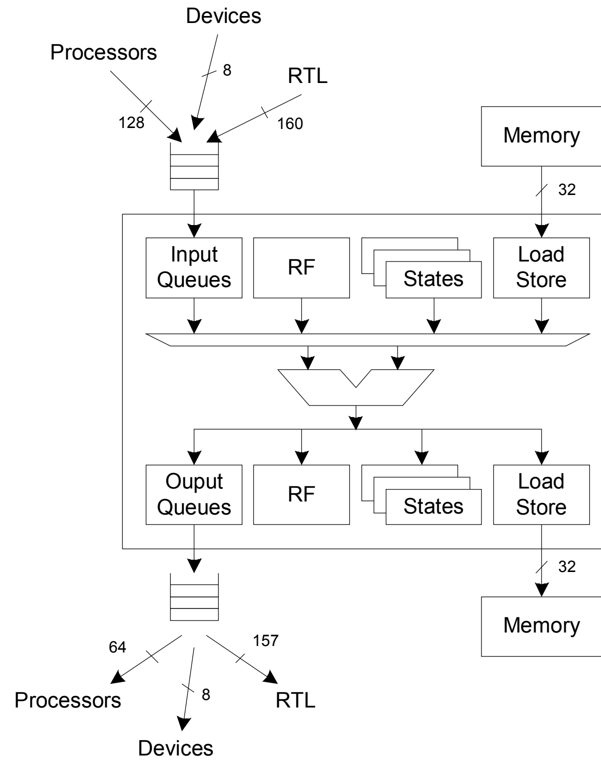

Figure 9: TIE queues used as instruction operands

The Xtensa LX processor itself includes 2-entry buffering for of every TIE queue defined by the system designer. The area consumed by a queue's 2-entry buffer is substantially smaller than a load/store unit. Thus the processor area consumed by TIE queue interface ports is all under the designer's direct control, and can be as small or large as necessary.

Flow-Through Processing

The availability of ports and Queues tied directly to a configurable processor's execution units permits the use of processors in an application domain previously reserved for hand-coded RTL logic blocks: flow-through processing.

By combining input and output queue interfaces with designer-defined execution units, it's possible to create a firmware-controlled processing block within a processor that can read values from input queues, perform a computation on those values, and output the result of that computation with a pipelined throughput of one clock per complete input-compute-output cycle.

Figure 10 illustrates a simple design of such a system with two 256-bit input queues, one 256-bit output queue, and a 256-bit adder/multiplexer execution unit. Although this processor extension runs under the control of firmware, its operation bypasses the processor's memory buses and load/store unit to achieve hardware-like processing speeds.

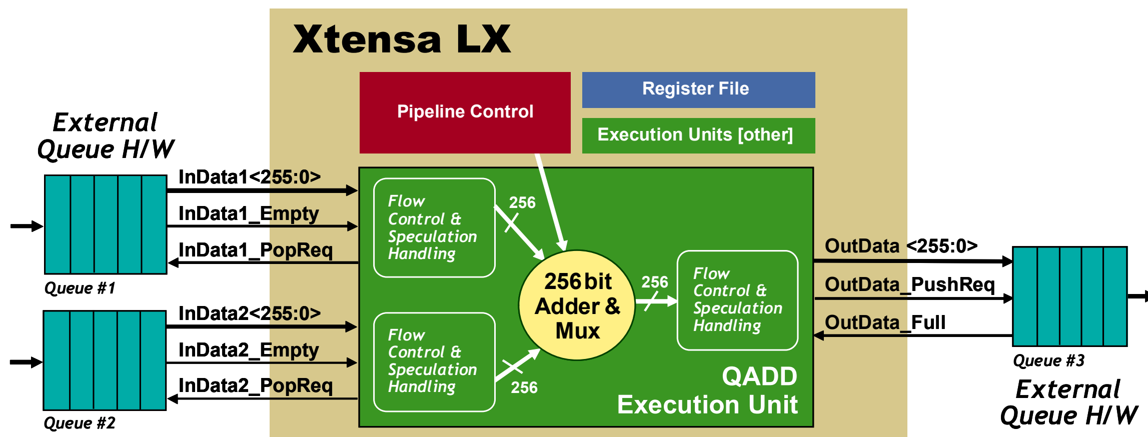

Figure 10: Combining queues with execution unit to add flow-through processing to a configurable processor core

Even though there is substantial hardware in this processor extension, its definition consumes only four lines of TIE code:

```
queue InData1 256 in
queue InData2 256 in
queue OutData 256 out
operation QADD {} { in InData1, in InData2, in SumCtrl, out
   OutData}    { assign OutData = SumCtrl ? (InData1 +
   InData2) : InData1;          }
```

The first three lines of this code define the 256-bit input and output queues and the fourth line defines a new processor instruction, QADD, which performs 256-bit additions or

passes 256-bit data from input to output. Defining the instruction in TIE tells the Xtensa Processor Generator to automatically add the appropriate hardware to the processor and to add the new instruction to the processor's software-development tool set.

Processor Cores for MPSOC Designs

The advent of the configurable processor core creates a new and very flexible building block for SOC designers. Configurable processor cores can achieve much higher performance than conventional, fixed-ISA processors through the addition of custom-tailored execution units, registers and register files, and specialized communication interface ports.

All of these innovations were impossible as long as processor design was shackled to the fixed-core mindset institutionalized more than 30 years ago with the introduction of the first microprocessor in 1971. For 21st century SOC design, these constraints no longer exist and it no longer makes sense to limit designers' use of processors with these obsolete constraints on system design.