Configurable Processors: A New Era in Chip Design

Configurable processors enable system-on-chip designers to leverage the benefits of nanometer silicon lithography with relatively little manual effort. These processors can achieve much higher performance than processors with conventional fixed-instruction-set architectures through the addition of custom-tailored execution units, registers, and register files as well as specialized communication interface ports.

Steve Leibson
James Kim
Tensilica

Microprocessor evolution can be broadly divided into three eras, each producing chips suited to their time. During the 1970s, microprocessors grew from 4-bit logic-replacement devices to 16- and 32-bit designs that paved the way for PCs and workstations. Explosive growth in 32-bit chips wiped out the minicomputer in the 1980s, which also saw the appearance of digital signal processors and other specialized architectures. Reduced-instruction-set computing dominated the 1990s; even stalwart complex-instruction-set computing cores such as the x86 evolved into disguised RISC architectures, and microprocessors became an integral part of mainframes and supercomputers.

Over the past three decades, the microprocessor has emerged as a fixed, stand-alone, reusable block created by highly skilled specialists. Because developing good, efficient microprocessor architectures can take years, many designers have come to regard them as monolithic entities subject to change only over long time periods and after careful consideration by an anointed few.

However, the rise of application-specific integrated circuit and system-on-chip (SoC) manufacturing technologies in the 1990s has laid the groundwork for a new, fourth era—that of post-RISC, configurable processors. Development tools are now advanced enough to allow any designer to tailor a microprocessor core for specific application tasks and to generate the processor's register transfer level (RTL) description plus all of the requisite software-development tools for that architecture in minutes, a shockingly brief time relative to the time spent designing processors and their associated development tools in prior eras.

Because of this ability to rapidly tailor processors for specific application tasks, configurable processors make excellent building blocks for SoC design, and developers can use them to quickly create functional blocks that might otherwise require months of manual labor to develop using handcrafted RTL descriptions. Consequently, various end products ranging from network routers to consumer electronics such as camcorders, printers, and video games already incorporate multiprocessor SoCs.

Two recent developments have further enmeshed configurable processors into SoC design:

- fully automated, application-specific instruction-set tailoring; and

0018-9162/05/$20.00 © 2005 IEEE

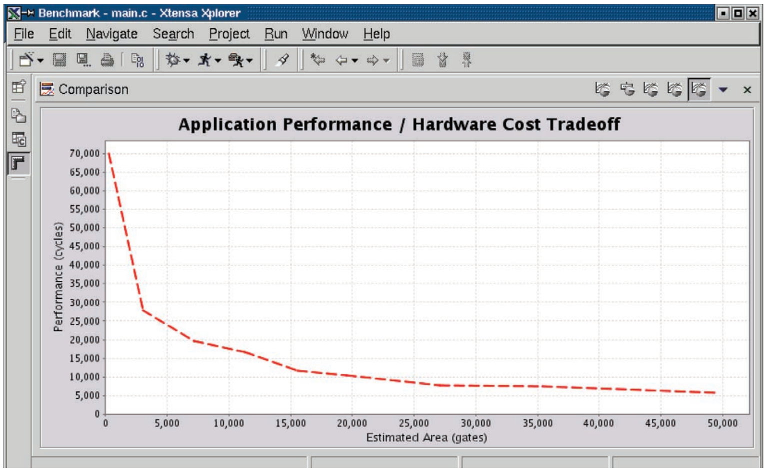
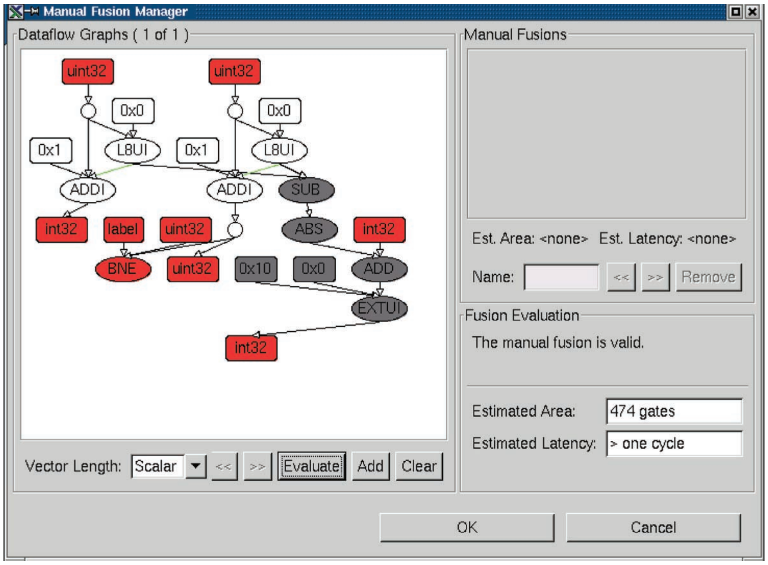


Figure 1. Xtensa processor extension synthesis compiler. XPRES creates a series of microprocessor configurations that provide increasing amounts of application-specific performance for an increasing amount of silicon area.



Est. Area: <none> Est. Latency: <none>

Name:

Fusion Evaluation

The manual fusion is valid.

Estimated Area: 474 gates

Estimated Latency: > one cycle

Vector Length: Scalar

Figure 2. XPRES dataflow graph with a series of operations marked as fusible. In this example, EXPRES estimates that a new instruction that fuses the subtraction, absolute-value, addition, and bit-field-extraction operations will require 474 additional gates.

- multiport access to the processor's internal execution units.

With automated tools for tailoring processors, SoC designers can focus more on system architectural issues to achieve performance goals rather than spending a lot of time on designing individual functional blocks within the SoC. Multiport access permanently shatters the formerly ironclad bus bottleneck that has choked microprocessor performance since the first commercial chip appeared in 1971.

AUTOMATIC PROCESSOR TAILORING

For more than a decade, hardware designers have struggled to transform system specifications written in C, and later C++, into efficient hardware. Developers often use these languages to write initial system or application specifications because they can execute and evaluate the specifications on inexpensive PCs. However, even PC hardware is too costly for many embedded systems designs, especially in the consumer electronics arena. Designers have thus continued looking for a tool that reduces executable specifications written in C or C++ to hardware.

Various approaches—including behavioral synthesis, C-language hardware synthesis, and electronic system-level design—have all fallen short of the mark because they try to solve an essentially intractable problem: transforming a description written in a sequentially executable language into a parallel collection of interoperating, nonprogrammable hardware blocks.

Tensilica's Xtensa processor extension synthesis (XPRES) compiler uses a simpler, more direct approach to tackle this problem. Instead of attempting to create application-specific hardware from scratch, XPRES starts with a fully functional microprocessor core (Xtensa), which can already run any C or C++ program, 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.

XPRES can search the available design space in less than an hour. This search results in a set of microprocessor configurations with a range of application performance and hardware cost characteristics (cost translates into silicon area on the SoC), as Figure 1 shows. The development team only needs to pick the configuration with the right performance/cost tradeoff for the target application and then submit it to Tensilica's Xtensa processor generator for implementation.

PERFORMANCE OPTIMIZATION

XPRES uses three techniques to create optimized Xtensa processor configurations: operator fusion; single instruction, multiple data (SIMD) vectorization; and flexible-length instruction extensions (FLIX).

Operator fusion

This technique notes the frequent occurrence of simple-operation sequences in program loops. XPRES combines these operation sequences into one enhanced instruction, which accelerates code execution by cutting the number of instructions executed within the loop, making the loop run faster, as well as reducing the number of instructions that must be fetched from memory, thus decreasing bus traffic. Figure 2 shows an XPRES-generated operation dataflow graph, with fusible operations marked in gray.

SIMD vectorization

Many loops within application programs repetitively perform the same operations on an array of data items. To vectorize such loops, XPRES creates an instruction with multiple identical execution units that operate on multiple data items in parallel. XPRES automatically tries two-, four-, and eight-operation SIMD vectorization in its design-space exploration. The addition of SIMD instructions to an Xtensa processor dovetails with Tensilica's Xtensa C/C++ (XCC) compiler, which has the ability to unroll and vectorize application programs' inner loops.

The loop acceleration achieved through vectorization is usually on the order of the number of SIMD units within the enhanced instruction. Thus, a two-operation SIMD instruction approximately doubles loop performance, and an eight-operation SIMD instruction speeds up loop execution by about a factor of eight.

FLIX

Unlike the multiple dependent operations of fused and SIMD instructions, FLIX instructions consist of multiple independent operations. Tensilica's XCC compiler can pack these operations into a FLIX-format instruction as needed to accelerate code. While fused and SIMD instructions are 24 bits wide, FLIX instructions are either 32 or 64 bits wide to allow the flexibility needed to fully describe multiple independent operations.

MULTIPLE CONFIGURABLE PROCESSORS

Few applications today can achieve their performance goals with a single processor, even with a configurable processor tailored to the target application's needs. However, the multiprocessor instruction sets, high-bandwidth interfaces, and small size of configurable processors encourage their extensive use in SoC designs. Advanced SoCs commonly use 10 or more configurable processors, and some high-end SoC designs use more than 100 complete processors per chip.

The choice of hardware-interconnection mechanisms among processor blocks in a SoC greatly affects performance and silicon cost, and these mechanisms must directly support the system design's interconnection requirements. Message-passing software communications naturally correspond to data queues, but message passing can be implemented using other types of hardware such as bus-based devices with global memory. Similarly, the shared-memory software-communications mode naturally corresponds to bus-based hardware, but techniques exist to physically implement shared-memory protocols even when no globally accessible physical memory exists. This implementation flexibility lets chip designers implement a spectrum of different task-to-task connections to optimize performance, power, and cost.

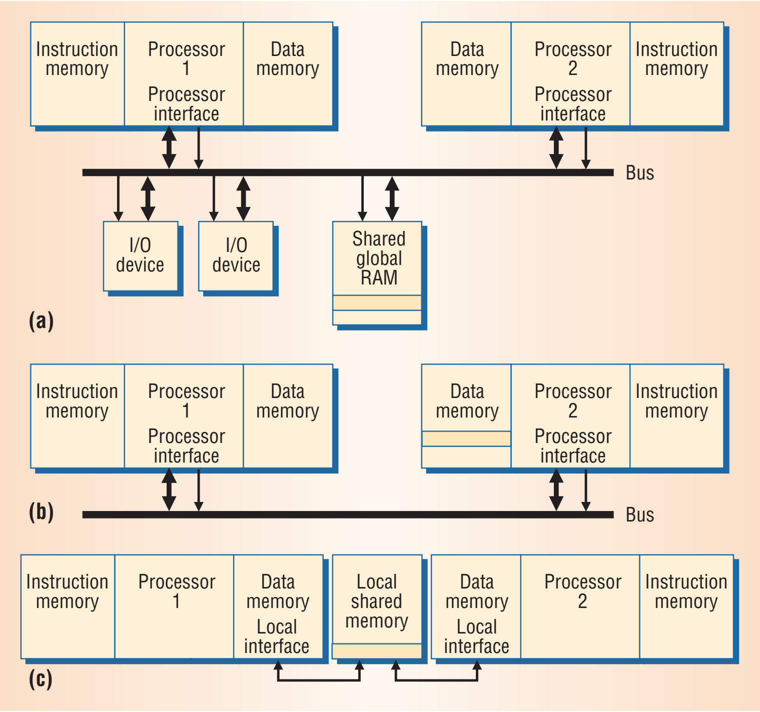


Figure 3. Shared-memory bus topologies using configurable processors. (a) Two processors access global memory over a bus. (b) One processor accesses the local data memory of a second processor over a bus. (c) Two processors share access to local data memory.

Configurable processors offer significant flexibility in supporting arbitrated access to shared devices and memory. The basic topologies for shared-memory buses are accessing remote global memory over a general processor bus, accessing local processor memory over a general processor bus, and accessing multiported local memory over a local bus.

Accessing global memory over a general bus

The processor can implement a general-purpose interface that allows a wide variety of bus transactions. If the processor determines that corresponding data is not local during a read, based on the target address or due to a cache miss, it must make a nonlocal reference over its main bus. The processor requests control of the bus, acquires bus control, and then sends the target read address over the bus. The appropriate device—for example, memory or an I/O interface—decodes that address and supplies the requested data back over the bus to the processor, as Figure 3a shows.

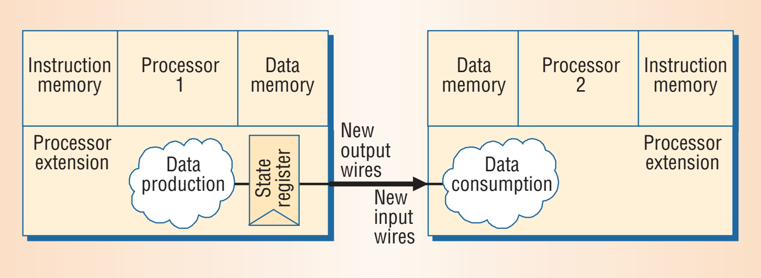


Figure 4. Direct processor-to-processor ports. Direct connection allows data to move directly from one processor's registers to the registers and execution units of another, reducing communication cost and latency.

When two processors communicate through global shared memory on the bus, one processor 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 I/O interfaces are accessible over a common bus. However, using global memory is inefficient and does not scale well with the number of processors and devices because increased bus traffic leads to long and unpredictable contention latency.

Accessing local memory over a general bus

Configurable processors can 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 serve as a bus slave and respond to requests on the general-purpose bus, as Figure 3b shows.

In this case, the read by processor 1 can require access arbitration when processor 1 requests access to the general-purpose bus and again when the read request reaches processor 2. The read request arrives over processor 2's interface and might contend with other requests for local-data-memory access from tasks running on processor 2. These two levels of arbitration can increase the access latency that processor 1 encounters, but processor 2 avoids access latency almost entirely because latency to local data memory is short—usually one or two cycles.

This latency asymmetry between processors 1 and 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. Processor 2 sees minimal latency when it reads the data because the data is local. Similarly, when processor 2 wants to send data back to processor 1, it writes the data into processor 1's local data memory.

Accessing multiported local memory over a 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 intertask communications. Each processor uses its local-data-memory interface to access shared memory, as Figure 3c shows. This memory could have two physical access ports (which can handle two memory references each cycle), or it could be controlled by a simple arbiter that holds off one processor's access for a cycle while the other processor is using the single physical access port.

Arbitration for a single port is preferred in area- and cost-sensitive applications, especially when shared-memory utilization is modest, because a true dual-ported memory is about twice as big per bit as single-ported RAM. However, true dual-ported memory may be preferable when the shared memory is very small or when the application requires absolute determinism of access latency.

DIRECT-CONNECT PORTS

Direct processor-to-processor connections reduce communication cost and latency by allowing data to move directly from one processor's registers to the registers and execution units of another. Figure 4 shows a simple example of direct connection. This example exploits the exporting of register state and imported wire values—features found in some extensible processors—to create an additional dedicated interface within each processor and to directly connect them.

When processor 1 writes a value to the output register, usually as part of some computation, that value automatically appears on the processor's output port. That same value is immediately available as an input value to operations in processor 2. Wire connections can be arbitrarily wide, allowing the quick and easy transfer of large and non-power-of-two-sized operands.

The operation that produces data for the output state register can be a simple register-to-register transfer or a complex logic function based on many other processor state values. Similarly, the receiving processor can simply transfer the input value to a processor state (register or memory) within itself, or it could use the value as one input to a complex logic function.

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) because queue widths need not be tied to a processor's bus width or general-register width. The handshake between data producer and consumer is implicit in the interfaces between the processors and the queue's head and tail.

Push and pop operations

The data producer pushes the data into the tail of the queue, assuming the queue is not full; if the queue is full, the producer stalls. When ready, the data consumer pops data from the head of the queue, assuming the queue is not empty; if the queue is empty, the consumer stalls.

The SoC designer also can create nonblocking push and pop operations for queues. Such queue operations in the data producer explicitly check for a full queue before attempting a push. The data consumer can explicitly check for an empty queue before attempting a pop. These mechanisms let the data producer or consumer move to other work in lieu of stalling.

Application-specific processors' instruction-set extensions allow direct implementation of queues. An instruction can specify a queue as one of the destinations for result values or use an incoming queue value as one operand source. Such operations can create a new data value or use an incoming data value during each cycle on each queue interface.

As Figure 5 shows, a complex processor extension can perform multiple queue operations per cycle, combining input from two input queues with local data and sending values to two output queues. A queue's high aggregate bandwidth and low control overhead enable using application-specific processors for applications with very high data rates, which processors with conventional bus or memory interfaces cannot handle.

Queue sizing

Queues decouple the performance of one task from another. If the data production and consumption rates are uniform, the queue can be shallow. If either the production or consumption rate is highly variable, a deep queue can mask this mismatch and ensure throughput at the average data producer and consumer rates, rather than at their minimum rate.

Queue sizing is an important optimization driven by good system-level simulation. If the queue is too shallow, the processor at one end of the communication channel can stall when the other processor slows for some reason; if the queue is too deep, the silicon cost will be excessive.

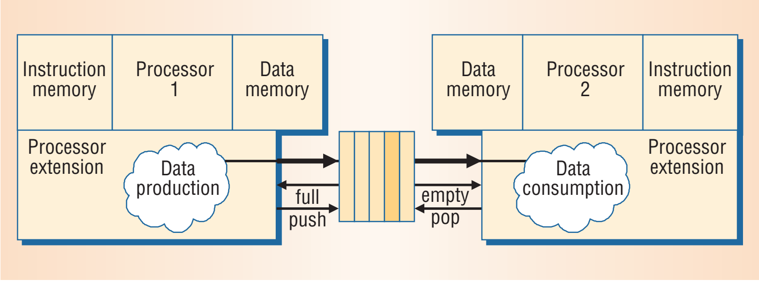


Figure 5. Data-queue mechanism. A complex processor extension can perform multiple queue operations per cycle, combining input from two input queues with local data and sending values to two output queues.

Queue interfaces

Queue interfaces to processor execution units are an unusual feature of commercial microprocessor cores. They become part of an Xtensa LX processor through the Tensilica instruction extension (TIE) language syntax, which defines the queue's name, width, and direction:

```
queue <queue-name> <width> inlout
```

One Xtensa LX processor can have more than 300 queue interfaces of variable width up to 1,024 bits each. These limits are set beyond the routing limits of current silicon technology so that the processor core's architecture is not the limiting factor in a system's design. The designer sets the practical limit based on system requirements, computer-aided design flow, and process technology selection. Using queues, designers can trade off fast and narrow processor interfaces with slower and wider interfaces to achieve bandwidth, performance, and power goals.

Figure 6 shows how TIE queues easily connect to simple DesignWare first-in, first-out memories. FIFO empty and full status signals gate TIE queue push and pop requests to comply with the DesignWare FIFO specification. The diagn_input is driven high, and the almost_full, half_full, almost_empty, and error outputs are unused. More elaborate FIFO memory implementations might be able to exploit the request signals when FIFO memory is nearly empty or full.

TIE queues serve directly as input and output operands of TIE instructions, just like a register operand, state, or 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;
}
```

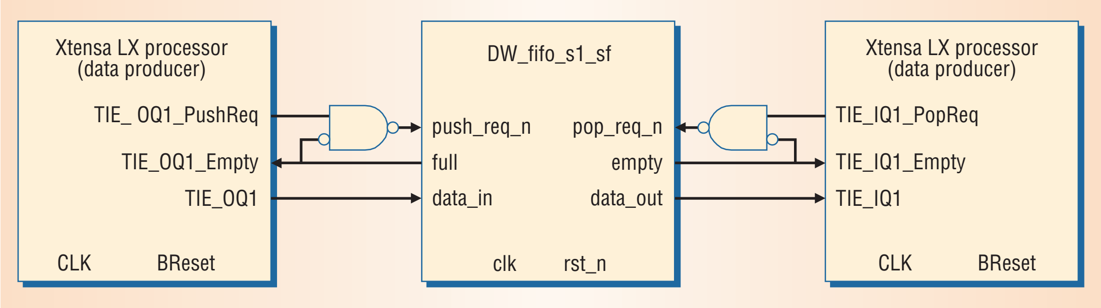


Figure 6. DesignWare synchronous first-in, first-out controller used with TIE queues. FIFO buffering provides a registered and synchronous interface to the external agent. Two entries buffer the processor from stalls when the attached external FIFO controller is full. In addition, buffering hides the processor's speculative execution from the external FIFO controller.

Figure 7 shows how TIE queues can function just like other instruction operands in an Xtensa LX processor. The figure also illustrates a key difference between queue interfaces and memory interfaces: The system designer can customize the width of each queue interface port to the exact value desired, either wider or narrower than the processor's standard memory interface ports.

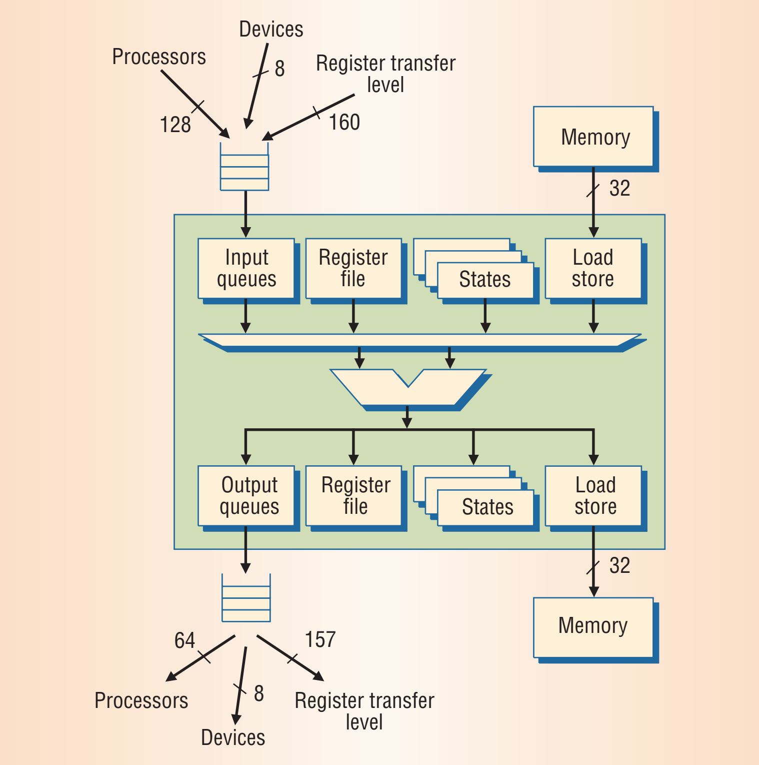


Figure 7. TIE queues used as instruction operands. The system designer can customize each queue interface port to the exact value desired, either wider or narrower than the processor's standard memory interface ports.

Queue buffering

Whereas memory accesses often exploit temporal locality, queue data is naturally transient. Consequently, queue storage can typically be smaller than a general-purpose memory buffer used for similar purposes. The Xtensa LX processor includes two-entry buffering for every TIE queue interface that the system designer defines. A queue interface's two-entry buffer consumes a substantially smaller area than a memory load/store unit, which can have large combinational blocks for alignment, rotation, and sign extension of data as well as cache-line buffers, write buffers, and complicated state machines. Thus, the processor area that TIE queue interface ports consume is under the designer's direct control and can be quite small or as large as necessary.

The FIFO buffering incorporated into the Xtensa LX processor for TIE queues serves three distinct purposes. First, it provides a registered and synchronous interface to the external agent (the actual FIFO memory), which portable IP blocks need to meet timing requirements under widely varying uses. Second, for output queues, buffering provides two entries that buffer the processor from stalls when the attached external FIFO memory is full. Third, it hides the processor's speculative instruction executions from the external FIFO memory.

HANDLING SPECULATION

Speculative loads occur on input queue interfaces because instructions operate on the queue data before these instructions are guaranteed to have completed all operations (before they reach the processor's commit stage). Activating a queue interface after the commit stage could be nonspeculative, but it would also be less useful, because a subsequent instruction that operated on that queue

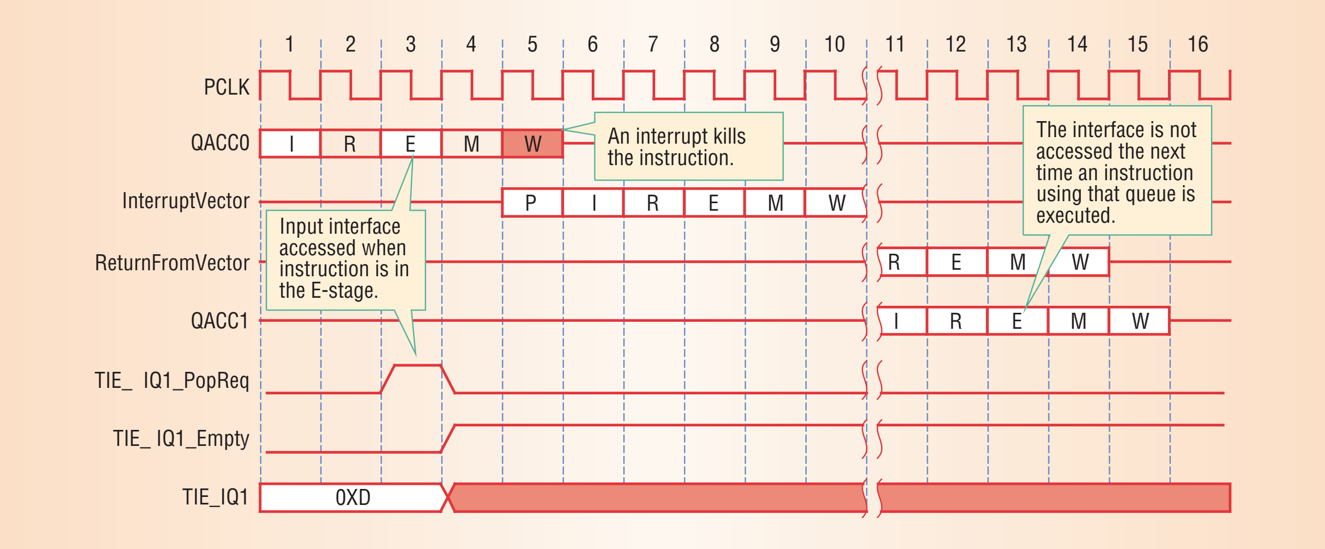

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
PCLK
QACC0
I
R
E
M
W
An interrupt kills the instruction.
InterruptVector
P
I
R
E
M
W
Input interface accessed when instruction is in the E-stage.
ReturnFromVector
R
E
M
W
The interface is not accessed the next time an instruction using that queue is executed.
QACC1
I
R
E
M
W
TIE_ IQ1_PopReq
TIE_ IQ1_Empty
TIE_IQ1
0XD

Figure 8. Speculative buffering. The processor handles speculative loads from a FIFO controller by keeping a temporary copy of data read from an input interface in a special buffer.

data would have to wait several cycles for the read-after-write hazard to resolve.

Speculative buffering

The processor can handle speculative loads from FIFO memory more effectively via *speculative buffering*, as Figure 8 shows. As an instruction reads data to be used as an operand from a TIE queue input interface, the queue's dedicated speculative buffer stores a copy of that data. The speculative buffer frees this entry only when the instruction commits. If the instruction does not commit—due to an exception, interrupt, or branch—the queue data remains in the buffer until the processor executes the next queue-reading instruction. This second execution obtains the internally buffered data, rather than reading a new value from the external FIFO memory, thereby preserving the ordering and coherence of queue references.

Figure 9 shows an example of speculative buffer timing. In cycle 3, the QACC0 instruction reads the input queue IQ1. Before the instruction can commit, an interrupt in cycle 5 kills it. The next instruction to execute that reads the input queue IQ1 is the QACC1 instruction. When this instruction executes, it uses the buffered queue data rather than issuing a new pop request to the associated FIFO memory in cycle 13.

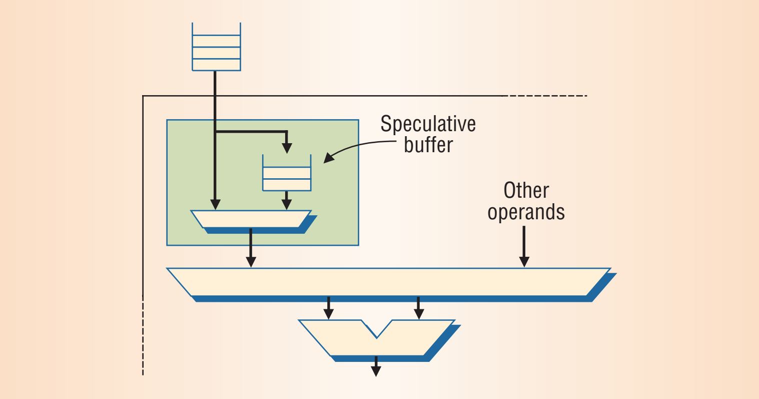


Figure 9. Example of speculative buffer timing. If the input queue indicates that it is not ready by asserting TIE_IQ1_Empty, by default, a queue is blocked, and processor execution stalls until it becomes available.

Speculative writes to output queues

Speculative writes to output queues are much simpler and work similarly to speculative writes to the processor's register files and states. That is, writes to output queues are only visible outside of the processor when the instruction commits. Speculation within the processor is handled by pipelining results to the commit stage.

For example, in Figure 9, if the input queue indicates that it is not ready by asserting TIE_IQ1_Empty, by default, the queue is blocked and processor execution stalls until data becomes available. The same would be true for a write to a full output queue. This hardware blocking mechanism permits a simple and straightforward approach to synchronization between a producer and a consumer.

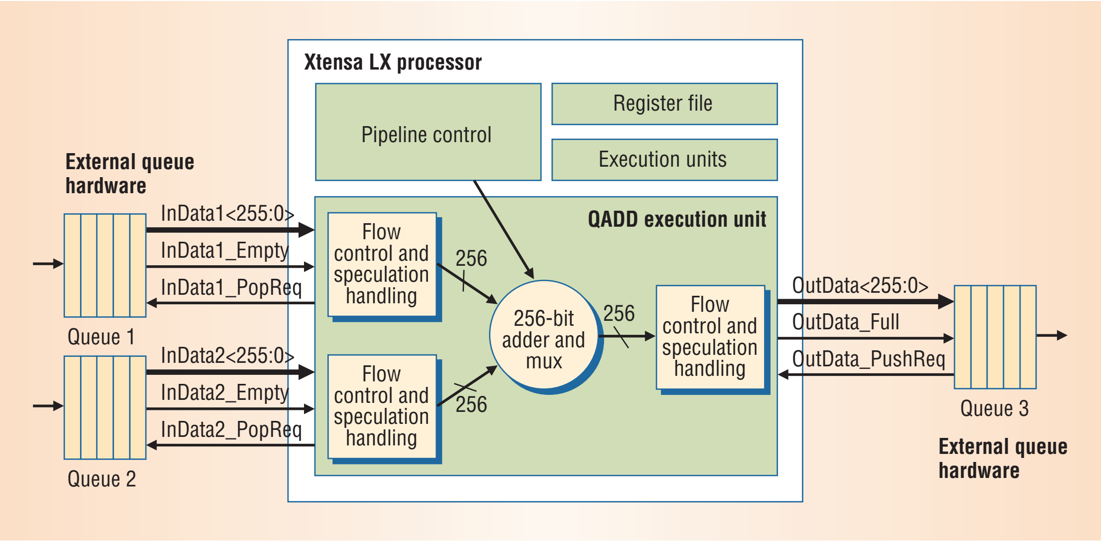


Figure 10. Flow-through processing. *Combining queues with execution units adds flow-through processing to a configurable processor core.*

In contrast, synchronizing communications using a shared-memory model is accomplished through semaphores and synchronization primitives, which is far more complicated. First, semaphores and data must use separate address spaces. The producer and consumer both poll the semaphore location to read each other's status. In addition, the data producer's synchronization software must guarantee the ordering of writes to the semaphore relative to writes to the data array to ensure the data consumer does not read the updated semaphore before all writes to data memory have completed.

There are many different approaches to memory ordering and synchronization, but all this effort is unnecessary when using a queue implementation that employs built-in hardware synchronization through FIFO memory's empty and full mechanisms.

Nonblocking queue accesses

Using nonblocking code for queue accesses is preferable when other tasks or processes can execute concurrently on the processor and when queue stalls can take many cycles. During these stalls it would be useful to switch to another task and return to the current thread later when the queue becomes available.

Code running on an Xtensa LX processor can perform nonblocking queue accesses by explicitly checking the queue's status and branching before executing the queue instruction itself, as shown in the following code snippets.
Assembly code for nonblocking queue access:

```
TASKA:
  check_queue_full b1  // queue status
    assigned to bool
  bnez b1, TASKB       // switch tasks if
    queue is full
  write_queue a1       // write to queue
       [...]
TASKB:
```

C for nonblocking queue access:

```
if(!check_queue_full()) {
    write_queue(value);
} else {
    [task b]
}
```

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*. Combining input and output queue interfaces with designer-defined execution units makes it 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 results with a pipelined throughput of one complete input-compute-output cycle per clock.

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 (mux) execution unit. Although this processor extension runs under firmware control, its operation bypasses the processor's memory buses and load/store unit to achieve hardware-like processing speeds.

Despite 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 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.

Fixed-core processors with a fixed instruction set and limited numbers of I/O ports and load/store units were appropriate in the days when microprocessors came in pin-limited packages, software development tools were handcrafted over a period of months or years, and system designs were undertaken at the board level. However, for 21st-century SoC design, such constraints are obsolete.

The configurable processor represents the next evolutionary step in microprocessor development, paving the way for many new and interesting system architectures that employ multiple, heterogeneous processor cores and exploit the qualities of advanced semiconductor lithography. Configurable processors provide SoC designers with building blocks that achieve performance rivaling hand-built RTL hardware blocks with postfabrication flexibility and firmware programmability but with much lower block-development and verification costs. ■

***Steve Leibson** is technology evangelist for Tensilica Inc. (www.tensilica.com), based in Santa Clara, California. His research interests include processor architectures and advanced system-level design. Leibson received a BSEE from Case Western Reserve University. He is a senior member of the IEEE and a 30-year member of the IEEE Computer Society. Contact him at sleibson@tensilica.com.*

***James Kim** is a senior design engineer in the hardware group at Tensilica Inc. His research interests include high-bandwidth processor interfaces and low-power design. Kim received an MSEE from Stanford University. Contact him at jameskim@tensilica.com.*