cādence®

High-Speed Alternatives for Inter-Processor Communications in SOCs

A flexible approach to communication techniques—rather than a typical global bus or bus hierarchy solution—enables ASIC and system-on-chip (SOC) designer teams to create cost-effective, task-to-task connection solutions that optimize performance, energy, and cost. This white paper presents some of the most common hardware mechanisms used to interconnect processor IP: buses and direct connections for Ports (GPIO), Queues (FIFOs), and memory Lookup interfaces using the Cadence® Tensilica® Xtensa® customizable processors.

Contents

Introduction.................................1

Processor Buses 2

Bus Design Tradeoffs 2

Bus Implementation with Xtensa Customizable Processors..............3

Direct Connections to Xtensa Processor Ports, Queues, and Lookups...............................4

Using Ports (GPIO).......................5

Using a DMA Controller to Manage Data Transfers over a System Bus 7

Using Queues (FIFO Interfaces).....7

Using Lookup Interfaces............. 10

Conclusion................................ 11

Additional Information............... 11

Introduction

The choice of hardware-interconnection mechanisms among processor blocks in a system on chip (SOC) affects communication performance and silicon cost. The default on-chip communications choice for most ASIC and SOC design teams—the global bus or a bus hierarchy—automatically incurs many performance and design problems. Other choices frequently used by software developers such as the message-passing software communications approach are commonly implemented using data queues. However, message passing can also be implemented using other types of hardware such as bus-based hardware with global memory. Additionally, the shared-memory software-communications approach works well with bus-based hardware. Note that shared-memory protocols can be physically implemented even when no globally accessible physical memory exists. Flexibility when implementing on-chip communications allows chip designers to implement a spectrum of different task-to-task connections in ways that optimize performance, power, and cost.

This white paper analyzes the effectiveness of the most common hardware mechanisms—buses and direct connections—used to interconnect processors on ASICs and SOCs using the Xtensa customizable processors. Except where explicitly noted, this paper assumes a one-to-one correspondence between tasks and processors. In fact, multiple tasks can be mapped onto one time-sliced processor and some tasks can be implemented by other non-programmable hardware accelerator blocks.

Processor Buses

A bus is a shared-access hardware mechanism allowing one or more processors to communicate with slave memories and I/O blocks. In the simplest system designs, each slave is accessible only from one bus. The processor that owns that bus also owns the slaves. In bus-based, multiprocessor systems, different processors must arbitrate for the bus, but this is the sole arbitration mechanism. Processors and slaves can have a range of bus-transfer requirements or traffic patterns, based on hardware limitations. For example:

- An 8-bit UART slave device does not allow any 16- or 32-bit transfers (a bus-transfer limitation)
- The processor may maximize performance with cache-line-sized block transfers—16 bytes or more (a traffic-pattern limitation)

Moreover, some transfers may be quite sensitive to latency (i.e., the task may not need much data but when it needs data, it needs that data immediately), and others may be more sensitive to bandwidth (i.e., the task requires an average sustained bandwidth, but the latency of any one transfer is inconsequential).

Bus Design Tradeoffs

Bus-centric system designs use a range of strategies and design decisions to satisfy conflicting goals among the processors, memories, and other devices.

Bus width and clock rate

The bus width and clock rate determine the peak transfer rate over the bus. These factors affect cost, power, and technology requirements.

Arbitration

The arbitration mechanism affects tradeoffs between bus utilization and the latency seen by any one bus master.

- **Round-robin arbitration** gives all masters equal access to the bus, but even the most important requests can face long contention delays. This type of arbitration is not only fair—all masters have an equal chance to get control of the bus—but it is also efficient because bus cycles are consumed only if a master needs them.
- **Strict-priority arbitration** gives the most critical bus master preferential treatment all the time so that it sees minimum contention latency.
- **Reserved-bandwidth arbitration** gives a bus master a minimum guaranteed bandwidth over a time interval, but the master can also compete for additional bandwidth on a round-robin basis.

The choice of arbitration mechanism is driven by the system bandwidth and latency requirements, but both bandwidth and latency can be constrained by pre-defined bus protocols.

Transfer types

Simple buses often implement just a few transfer types such as 1- (high-speed serial communication), 8-, 16-, and 32-bit reads and writes. More complex buses implement more advanced transfer types, including:

- **Fixed-block transfers**—Power-of-two sized block transfers, often used for cache-line refills and write-backs.
- **Variable-block transfers**—Arbitrary-length block transfers, often used to move streamed data with application-dependent block sizes.
- **Split transactions**—The decomposition of a bus request (usually a read) into two transfers: one to convey an address from the master to the slave, and a second to return a response data block from the slave to the master. The bus is relinquished to other masters during the interval between the request and response. Split transactions are particularly important for maintaining high bus bandwidth with long memory device latencies and multiple bus masters.
- **Atomic transactions**—In some scenarios where two or more masters compete for access to a shared resource, a locking mechanism (for a short-time duration) can be used to support resource-arbitration mechanisms. Sometimes this mechanism is implemented as a bus lock, in which certain read operations retain bus mastership after the read data is returned so that the processor can perform a write without risk that another processor may read the same location.

Bus Implementation with Xtensa Customizable Processors

The Xtensa customizable processors offer significant flexibility in supporting arbitrated access to shared devices and memory over a system bus. The basic topologies for shared memory buses are listed below.

Global memory access over a system bus

The processor is attached to a system bus that allows a wide variety of transactions. If the processor determines that the corresponding data is not local during a read based on the data's address or due to a cache miss, the processor must make a non-local reference. As a result, 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, a memory or I/O device) decodes that address and supplies the requested data back over the bus to the processor, as shown in Figure 1.

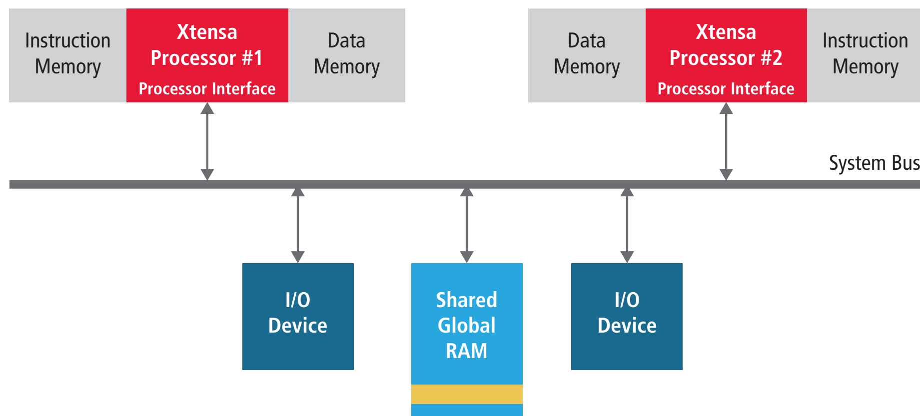

Figure 1: Two Processors Access Shared Memory over a System Bus

When two processors communicate through global shared memory on the bus, one processor must acquire bus control to write the data and the other processor must later acquire bus control to read the data. Each word transferred in this fashion therefore requires two bus transactions. This approach requires modest hardware and is very flexible, because the global memories and I/O blocks 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. Bus-arbitration overhead becomes significant if many processors must contend for memory or I/O access, which reduces the maximum available bus throughput.

Local memory of another processor accessed over a system bus

Processors can have local data memories that participate in general-purpose bus transactions. Additionally, an appropriately configured processor can also serve as a bus slave and can respond to access requests that target its local memory on the system bus, as shown in Figure 2.

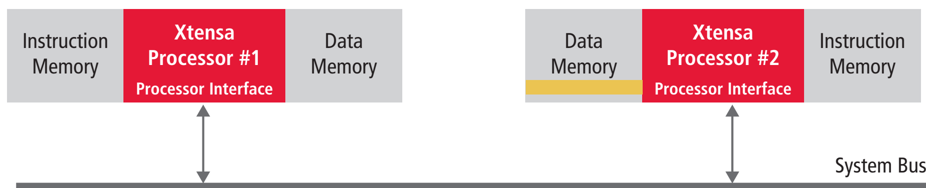

Figure 2: One Processor Accesses the Local Data Memory of a Second Processor over a System Bus

In this case, the read by Processor 1 can require access arbitration at two levels:

- Processor 1 must request access to the system bus.
- When the read request reaches Processor 2, it must contend with other requests for local data-memory access from tasks running on Processor 2. Therefore, the read request from Processor 1 must compete with Processor 2's local access requests.

The two arbitration levels can increase Processor 1's access latency, 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 Processor 1 and Processor 2 encourages push communication: when Processor 1 sends data to Processor 2, it writes the data over the system bus into Processor 2's local data memory. If the write is buffered—for example, in Processor 1's write buffer—Processor 1 can continue execution without waiting for the write to complete. Thus, the long latency of data transfer to Processor 2 is hidden from Processor 1. Processor 2 sees minimal latency when it writes the data to its local memory. Similarly, when Processor 2 wants to send data back to Processor 1, it writes the data into Processor 1's local data memory.

Multi-ported memory directly connected to processors

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 a local memory interface to access a shared memory, as shown in Figure 3. This memory can have two physical access ports (two memory accesses can be satisfied each cycle). Alternatively, a single memory interface can be controlled by a simple arbiter, where one processor's access request is held off for a cycle if the other processor is using the memory's single physical access port.

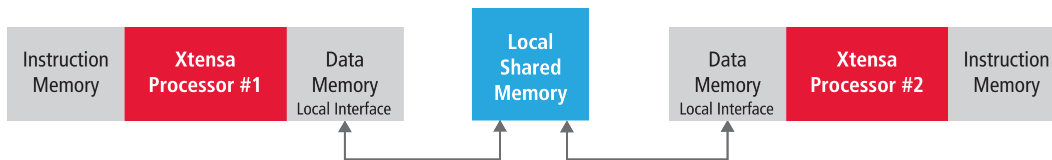

Figure 3: Two Processors Share Access to Local Data Memory

A true dual-ported memory is about twice as big per bit when compared to single-ported RAM, so a single port with access arbitration is preferred for area- and cost-sensitive applications, especially when shared-memory utilization is modest. However, a true dual-ported memory may be the better choice when the shared global memory is small or when absolute determinism is required for access latency.

Direct connections to local memories

The Xtensa processor offers several interface options for directly connecting to local memories (cache, RAM, ROM) for fast access to data.

See the *Xtensa Microprocessor Data Book* for details.

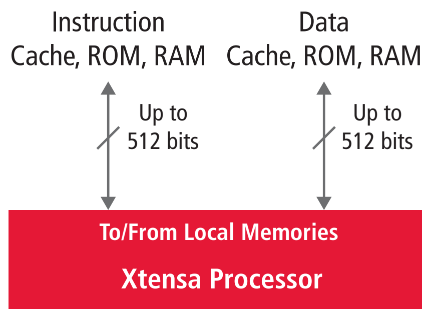

Figure 4: Processor Access to Local Memories

Direct Connections to Xtensa Processor Ports, Queues, and Lookups

Virtually every 32-bit processor on the market uses a system bus interface to transfer data to and from other parts of the SOC, which takes multiple clock cycle(s). Alternatively, the Xtensa customizable processor also allows you to entirely bypass the bus.

The Xtensa processor enables you to add multiple custom interfaces for directly connecting to external RTL blocks, FIFO data Queues, and memory blocks as shown in Figure 5. These Ports, Queue and memory Lookup interfaces can operate in parallel for very high throughput per cycle, and are independent of any local memory and system bus transfers as they are not mapped into the processor's memory address space.

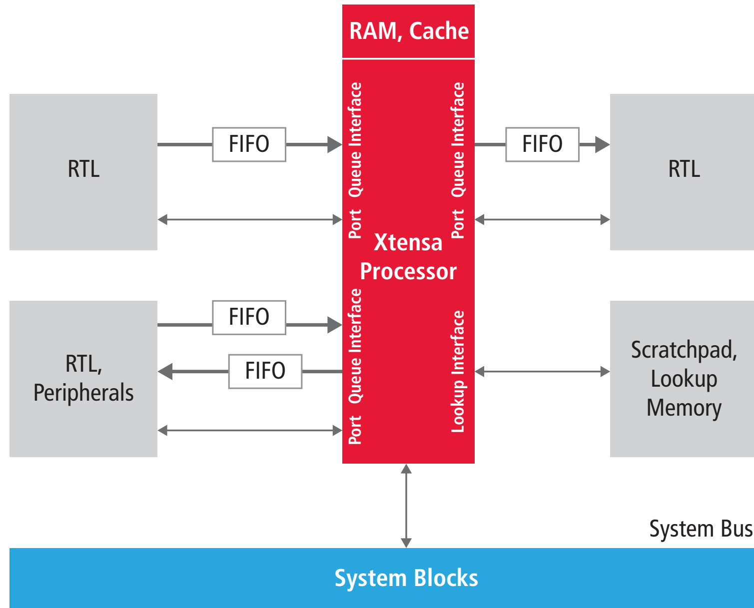

Figure 5: Xtensa Processor Ports, Queues, and Lookups

Using Ports (GPIO)

Direct processor-to-processor connections reduce the cost and latency of inter-processor communications. Direct connections allow data to move from one processor's registers to the registers and execution units of another processor without using a memory buffer for intermediate storage. Figure 6 shows a simple example of a direct processor-to-processor connection. This example takes advantage of special Port features found in Xtensa processors to create additional dedicated interfaces within the processor.

On the Xtensa processor, Ports act like general-purpose I/O (GPIO) and are wires that directly connect two Xtensa processors or an Xtensa processor to external RTL. Each Port connection can be up to 1,024 wires wide, allowing wide data types to be transferred easily without the need for multiple load/store operations. Ports are particularly useful to convey control and status information.

For example, when Processor 1 writes a value to an output Port, usually as part of some computation, that value automatically appears on that processor's output pins. That same value is immediately available on the input Port for use by operations in Processor 2. These Port connections can be arbitrarily wide, up to 1,024 bits, allowing large and non-power-of-two-sized operands to be transferred easily and quickly between processors.

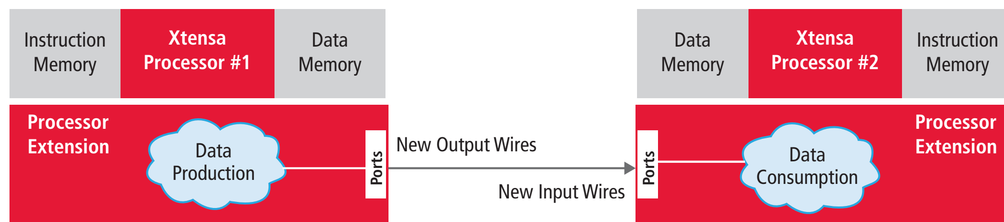

Figure 6: Direct Processor-to-Processor Ports

The operation that produces the data for the output Port state register can be as simple as a register-to-register transfer or it can be a complex logic function based on many other processor state values including inputs from other processor Port pins. Similarly, an input Port value can simply be transferred to another processor state within Processor 2 (register or memory), or used as one input to a complex logic function.

This form of direct connection still requires a handshake between the two processors. Processor 2 (the data consumer) can signal to Processor 1 (the data producer) that the data in the register has been used, so that the producer can write the next data value. The producer can signal to the consumer when the new data is available using one of several methods.

Consumer-to-producer port

A designer can create two additional Port connections in the processors, each just one bit wide. One Port connects the consumer processor back to the producer processor (an acknowledge signal), and the other connects the producer to the consumer (a data-ready signal). The producing processor asserts its “data-ready” handshake output when the data value is valid. The data-consuming processor asserts its “acknowledge” output pin when it has used or consumed the supplied data. The producer uses this acknowledgement as part of the decision tree in its firmware to generate the next output value. The producing processor negates its “data-ready” handshake output until the next data value is available. The consuming processor then negates its “acknowledge” signal to prepare for the next acknowledgement.

The corresponding handshake timing appears in Figure 7. Because this transaction requires at least one full instruction execution per signal transition, this method consumes at least a dozen cycles per data word transferred.

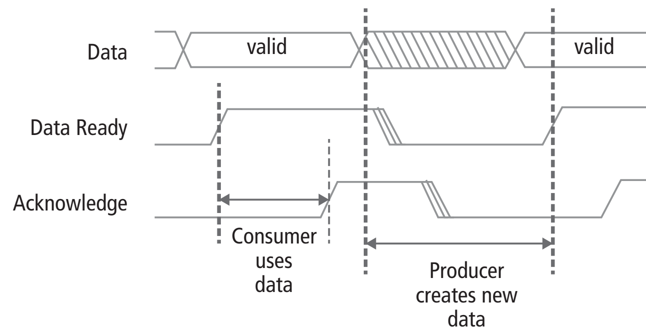

Figure 7: Two-Wire Handshake

The Xtensa processor’s output and input Ports can be used to integrate the processor tightly with peripherals. For example, these Ports can be used to send and receive control signals to and from other devices and RTL blocks.

Interrupt-driven handshake

The data transfer can also be controlled by interrupts exchanged between the two processors. The producing processor creates the data and places it on its output Port. It then asserts a signal on another output Port connected to an interrupt input of the consuming processor, which then handles the interrupt as soon as it can (after any higher priority interrupts are handled) and accepts the data from the input Port within the interrupt handler. The consuming processor’s interrupt handler then asserts its own output Port signal, which is connected to an interrupt input on the producing processor. The producing processor’s interrupt handler can then drive new data to the consumer. Figure 8 shows the basics of the interrupt-driven handshake.

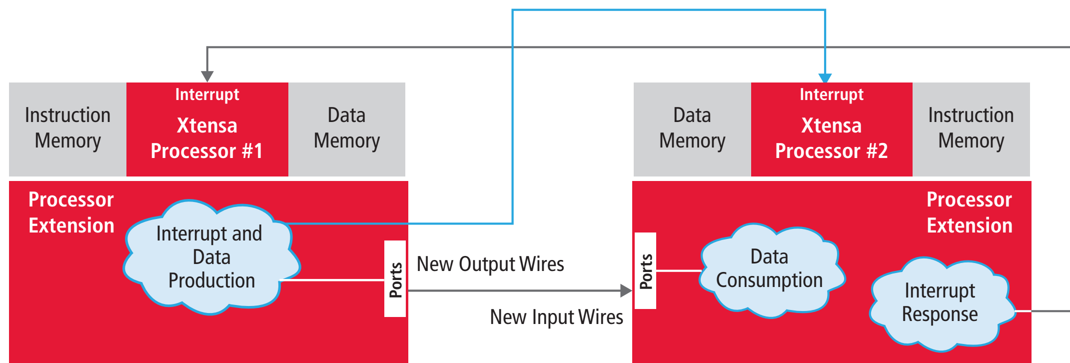

Figure 8: Interrupt-Driven Handshake

You can create messages for inter-core communication and synchronization of activities between the cores where each dedicates one interrupt pin and some shared memory. Using this approach, Processor 1 can offload several tasks to Processor 2 by sending different messages.

See the *Audio Codec Software Partitioning for a Two-Core System* application note for an implementation example using this approach.

Using a DMA Controller to Manage Data Transfers over a System Bus

To free up a processor to perform other processing tasks in parallel, a dedicated direct memory access (DMA) controller can be used to perform data transfers, or a DMA controller can be shared with another processor in a multi-processor environment where one processor offloads certain data transfer tasks to another. For an example, see the Audio Codec Software Partitioning for a Two-Core System application note.

Using Queues (FIFO Interfaces)

Queues (FIFO interfaces) provide a mechanism to buffer streaming data between two points. Input and output Queues are read and written as registers, but they are filled and emptied externally. Each Queue can transfer up to 1,024 bits every clock cycle.

The highest-bandwidth mechanism for task-to-task communication is implementing Queues in hardware. One Queue can sustain data rates as high as one transfer every cycle. This sort of interface is commonly used when designing hardware accelerator blocks, but it is uncommon for processor-based communications because most processors cannot directly support FIFO interfaces. However, with Queues Xtensa processors do support them and they are valuable when a designer needs to boost processor I/O transfer rates.

For the Xtensa processors, Queue widths need not be tied to a processor's bus width or general-purpose register width. As discussed in the previous section, the handshake between producer and consumer is implicit in the hardware interfaces between the processors and the Queue's head and tail. There is no associated software overhead with this sort of communication, which makes the technique extremely fast.

In operation, the data producer creates the data and pushes it into the tail of the Queue, assuming the Queue is not full. If the Queue is full, the producer stalls. When the data consumer is ready for new data, it pops a data word from the head of the Queue, assuming the Queue is not empty. If the Queue is empty, the consumer stalls. Stalls are automatically handled by processor hardware, which creates automatic data-flow control.

Queues can also be configured to provide non-blocking push and pop operations, where the producer can explicitly check for a full Queue before attempting a push and the consumer can explicitly check for an empty Queue before attempting a pop. This mechanism allows the producer or consumer tasks to move to other work in lieu of stalling the processor.

Xtensa-based 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 unidirectional Queue interface, like the one shown in Figure 9, allows data values of up to 1,024 bits in width to be created and consumed independently of each other. A single customized processor can support over 1,000 interfaces.

A complex processor extension can perform multiple Queue operations per cycle, perhaps combining inputs from two input Queues with local data and sending values to two output Queues. The high aggregate bandwidth and low control overhead of Queues allows Xtensa-based processors to be used for applications with high data rates where processors with conventional bus or memory interfaces are not appropriate because they cannot handle the required high data rates. In particular, Queues and Ports can be used to interface directly to custom RTL blocks elsewhere in the SOC.

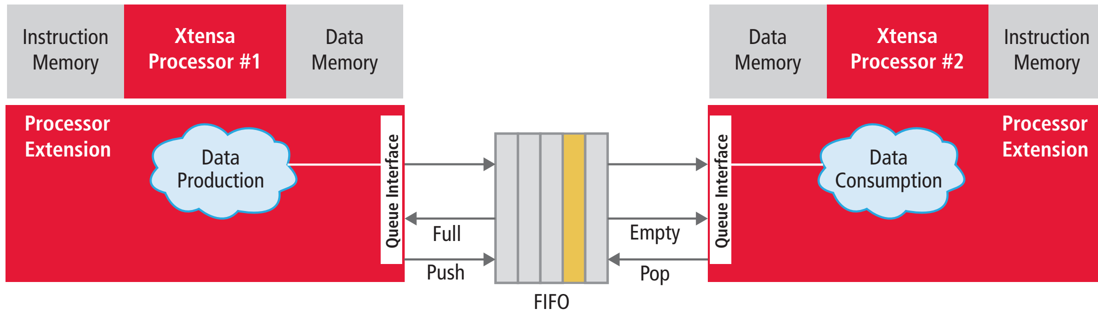

Figure 9: Processor to Processor Data Transfer using Queues

Queues decouple the performance of one task from another. If the data production and data consumption rates are uniform, the Queue FIFO can be shallow. If production or consumption rates are highly variable, a deep FIFO can mask this mismatch and ensure throughput at the average rate of producer and consumer, rather than at the minimum rate of the producer or consumer. Sizing the FIFOs is an important optimization and should be driven by system-level simulation. If the FIFO is too shallow, the processor at one end of the communication channel may stall when the other processor slows for some reason. If the FIFO is too deep, the silicon cost will be excessive.

One processor can implement Queue communications with multiple partners. When the Queue operations are directly incorporated into the instruction set, the code sequence entirely determines which Queue is written or read. Sometimes, less direct mapping is desirable, so the code sequence that produces or consumes data can be separated from the selection of the source or destination Queue.

Two methods for flexible Queue selection are possible. First, the ultimate data destination can be included in the data transfer. This destination information is pushed into a common Queue with the data. This Queue feeds other Queues, where simple logic pops the destination identifier and uses it to choose the correct Queue to receive the corresponding data. Flexible Queue widths make this approach economical. For example, a 2-bit destination specifier and a 32-bit data word can be combined into a 34-bit common Queue, perhaps feeding a set of four 32-bit Queues, as shown in Figure 10.

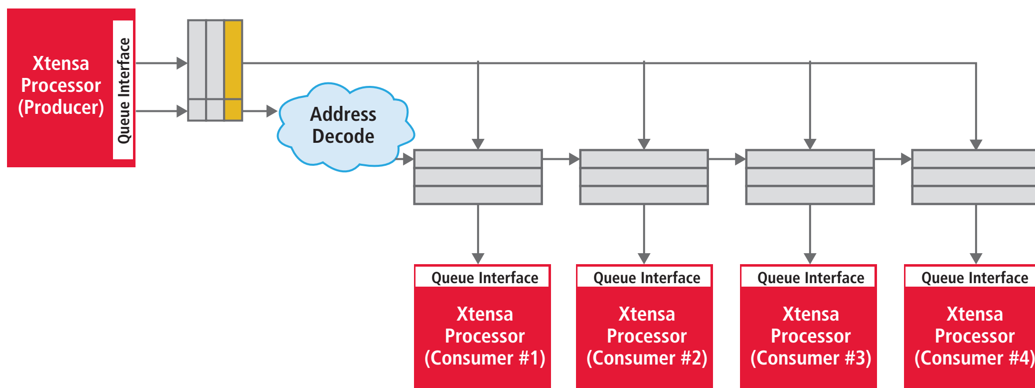

Figure 10: Producer Encodes Destination with Data

In an alternative approach, the Queue's head and tail can be memory-mapped, so that a processor's store operation pushes a data value onto the Queue and another processor's load operation pops the value from the Queue. These operations can be blocking (producing a stall if the Queue is full or empty) or non-blocking (processor tests the state of the Queue before attempting the push or pop).

Figure 11 shows a simple system with one producer and two consumers. The FIFO Queues are mapped into the address spaces of the processors, shown here using the local-memory space with a 1-cycle access time, so that a store to the address of the Queue's tail causes a push, and load from the address of the Queue's head causes a pop.

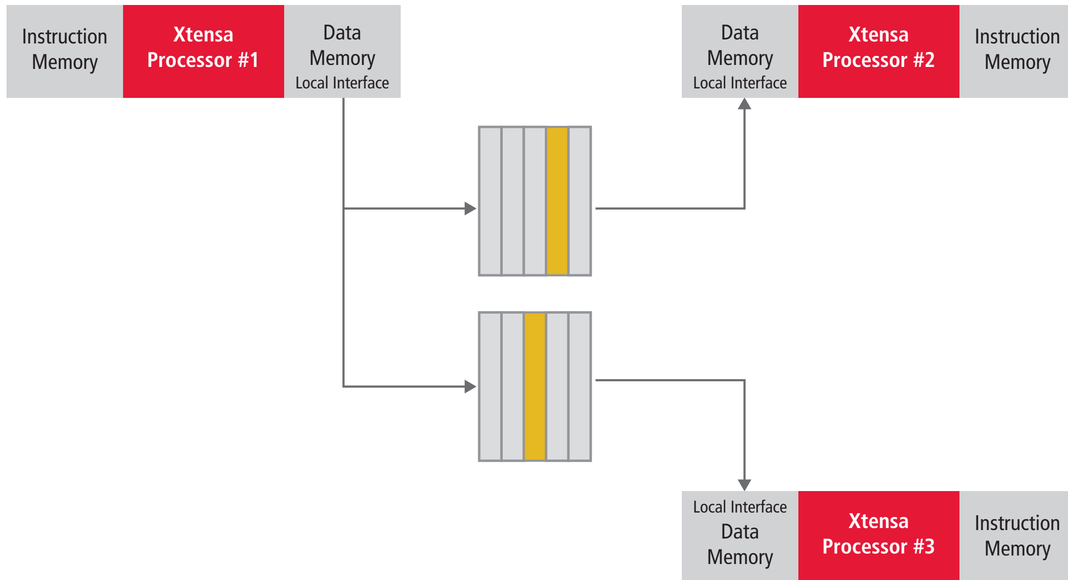

Figure 11: One Producer Serves Two Consumers through Memory-Mapped Queues

The FIFO depth can be small if the data rate is relatively low. It can even be a single entry—reduced to a register that is written by the producer and read by the consumer. This "mailbox" register serves as a simple and convenient path between producer and consumer. A memory-mapped set of mailbox registers is shown in Figure 12. When the two tasks pass data back and forth, the same register can be used for transfers in either direction.

Figure 12: Memory-mapped Mailbox Registers

Queues serve a wide range of processor communication uses. They work especially well at high data rates with relatively shallow buffering. At lower data rates, buses provide ample communications bandwidth. For applications with very deep buffering requirements, Queues can be implemented with sufficient FIFO RAM or replaced with a shared-memory communication mechanism.

Using Lookup Interfaces

Figure 13 shows the various Xtensa processor connection options to/from: TIE Ports, Queue and Lookup interfaces, local memories, and system bus. In particular, the lookup interfaces are useful for directly connecting external memories (e.g., ROM and RAM) as table Lookups or for connecting fixed-latency hardware computation units. Memories connected to Lookup interfaces are outside the normal processor memory map and are read and written directly from the processor using their own instructions. The Lookup interface can be used in parallel with transactions occurring on local memory interfaces.

Each device connected via a Lookup interface is accessed with a TIE instruction that can perform a transaction on one or more interfaces at once.

The Lookup interface is used to send a request (address + data) to an external device and then receive a response (data) within a pre-determined number of cycles. The request and response are treated as an atomic transaction within a TIE instruction.

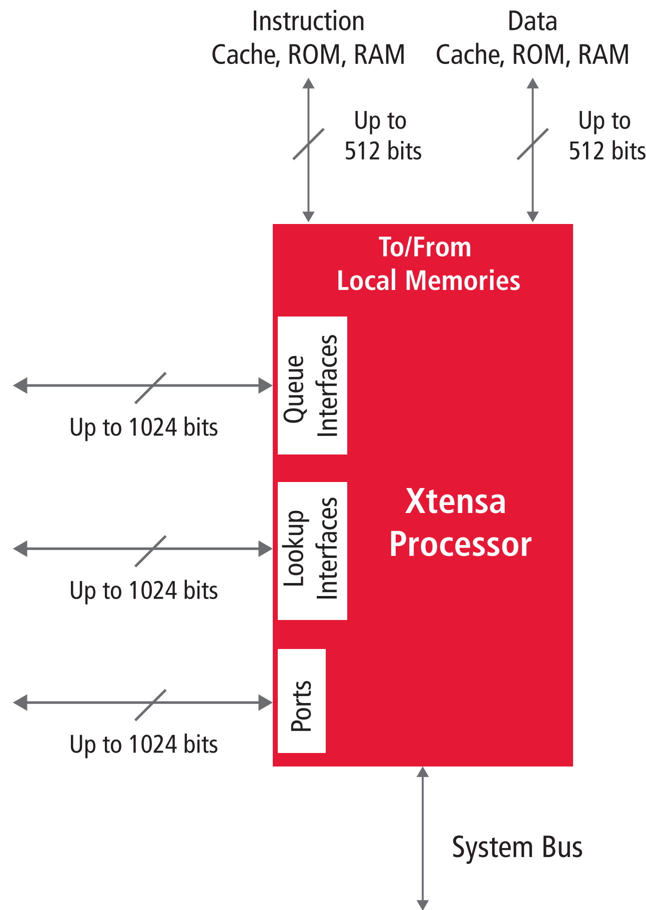

Figure 13: Xtensa Processor Connection Options

Conclusion

Limiting on-chip communications to global buses and bus hierarchies needlessly restricts on-chip communications bandwidth and increases the design effort needed to achieve all of the project's bandwidth and latency goals.

A broader view of the available communications techniques, such as the direct connections mentioned above that work well between processors and between processors and other RTL blocks will help ASIC and SOC design teams create more efficient and cost-effective designs in less time, with less effort, and with a lower risk of system failure.

Additional Information

For more information on the unique abilities and features of Cadence Tensilica Xtensa processors, see ip.cadence.com.

cadence®

Cadence Design Systems enables global electronic design innovation and plays an essential role in the creation of today's electronics. Customers use Cadence software, hardware, IP, and expertise to design and verify today's mobile, cloud and connectivity applications. www.cadence.com

© 2015 Cadence Design Systems, Inc. All rights reserved. Cadence, the Cadence logo, Tensilica, and Xtensa are registered trademarks of Cadence Design Systems, Inc. All others are properties of their respective holders. 4570 06/15 SC/DM/PDF