Get Your ASICs and SOCs Off the Bus!

High-Speed I/O Alternatives for Inter-Processor Communications on SOCs

The choice of hardware-interconnection mechanisms among processor blocks in an SOC affects communication performance and silicon cost. The default on-chip communications choice for most ASIC and SOC design teams is the global bus or a bus hierarchy, however this choice automatically incurs many performance and design problems. There are other choices that may be more appropriate for today's nanometer ASIC and SOC designs. These choices match well with communications concepts frequently used by software developers. For example, message-passing software communications have a natural correspondence to data queues. Message passing can be implemented using other types of hardware such as bus-based hardware with global memory. Similarly, the shared-memory software-communications mode has a natural correspondence to bus-based hardware, but 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 provides short descriptions of the most common hardware mechanisms—buses, direct connections, and data queues—used to interconnect processor cores on ASICs and SOCs. 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 may 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—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—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 to satisfy conflicting goals among the processors, memories, and other devices. Three classes of design decision stand out:

- **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. Round-robin arbitration is fair, in that all masters have an equal chance to get control of the bus, and it is efficient, in that bus cycles are consumed only if an 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 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:* When two or more masters compete for access to a shared resource, a locking mechanism is required 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 locking is not efficient in systems with many processors, many separate memories, and frequent locking operations.

Bus implementation with customizable processors

Tensilica's Xtensa customizable dataplane processors (DPUs) offer significant flexibility in supporting arbitrated access to shared devices and memory over buses. The basic topologies for shared memory buses are:

1. **Remote global memory accessed over a general processor bus:**

The processor is attached to a global bus that allows a wide variety of transactions. If the processor determines that 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 block) decodes that address and supplies the requested data back over the bus to the processor, as shown in Figure 1.

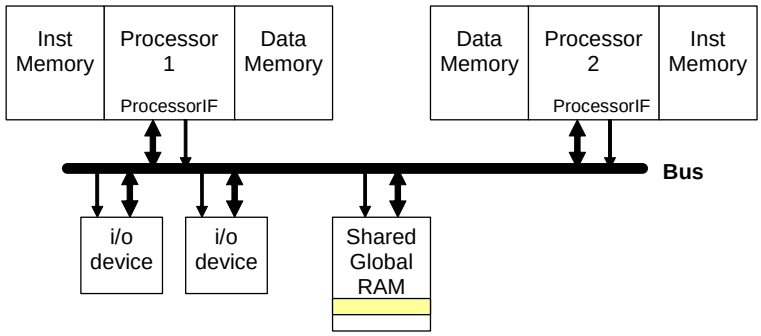

Figure 1: Two processors access shared memory over a bus

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 maintains high flexibility, 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.

2. Local processor memory accessed over a general processor bus:

DPUs can have local data memories that participate in general-purpose bus transactions. These data memories are primarily used by the processor to which they are closely coupled. However, an appropriately configured processor can serve as a bus slave and can respond to access requests that target its local memory on the general-purpose bus, as shown in Figure 2.

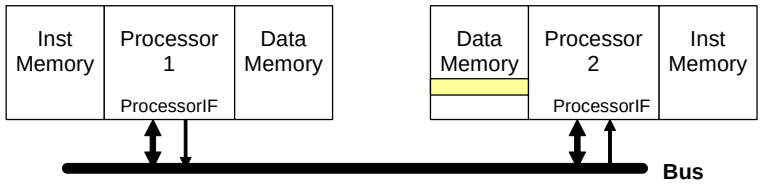

Figure 2: One processor accesses the local data memory of a second processor over a bus

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

- Processor 1 must request access to the general-purpose 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 the access latency seen by Processor 1 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 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.

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 3. This memory could have two physical access ports (two memory references can be satisfied each cycle) or a single memory interface could 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.

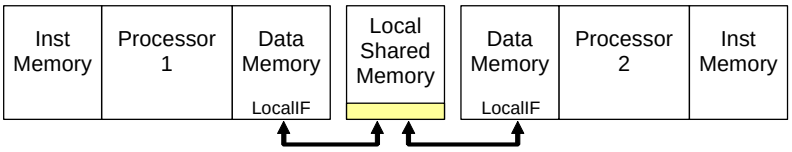

Figure 3: Two processors share access to local data memory

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 memory is very small or when absolute determinism is required for access latency.

Direct Connect Ports

Direct processor-to-processor connections reduce the cost and latency of inter-processor communications. Direct connections allow data to move directly from one processor's registers to the registers and execution units of another processor without using a memory buffer for intermediate storage. Figure 4 shows a simple example of a direct processor-to-processor connection. This example takes advantage of special port features found only in Tensilica's DPUs to create additional dedicated interfaces within the processor.

Whenever the Processor 1 writes a value to an internal output register, usually as part of some computation, that value automatically appears on that processor's configured output pins. That same value is immediately available as input value to operations in Processor 2 through the input port pins on that second DPU. Such port connections can be arbitrarily wide, much wider than 32 or 64 bits, allowing large and non-power-of-two-sized operands to be transferred easily and quickly between processors.

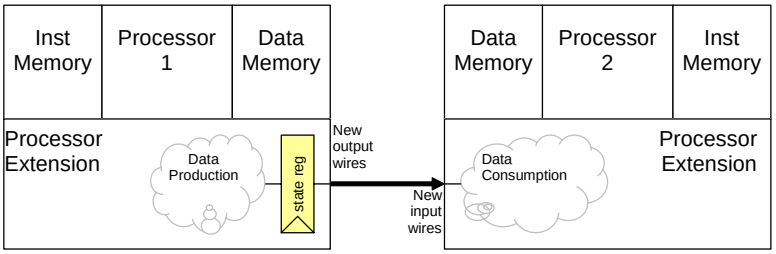

Figure 4: Direct processor-to-processor ports

Tensilica's DPUs allow you to create registers with exported state. The operation that produces the data for the output 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, the input value can simply be transferred to another processor state within Processor 2 (register or memory), or it could be used as one input to a complex logic function.

This form of direct connection still requires some sort of handshake between the two processors. The consumer of data may need to signal to the producer that the data in the register has been used, so that the producer can write the next data value. The producer may need to signal the consumer than new data is available.

This signaling can be done in several ways:

- **Consumer-to-producer port:** A designer can create two additional port connections in the DPUs, 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 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 then asserts its "data-ready" handshake output when the next data value is available. The consuming processor then negates its "acknowledge" signal, to prepare for the next acknowledgement, after it has processed the next data word.

  The corresponding handshake timing appears in Figure 5. 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.

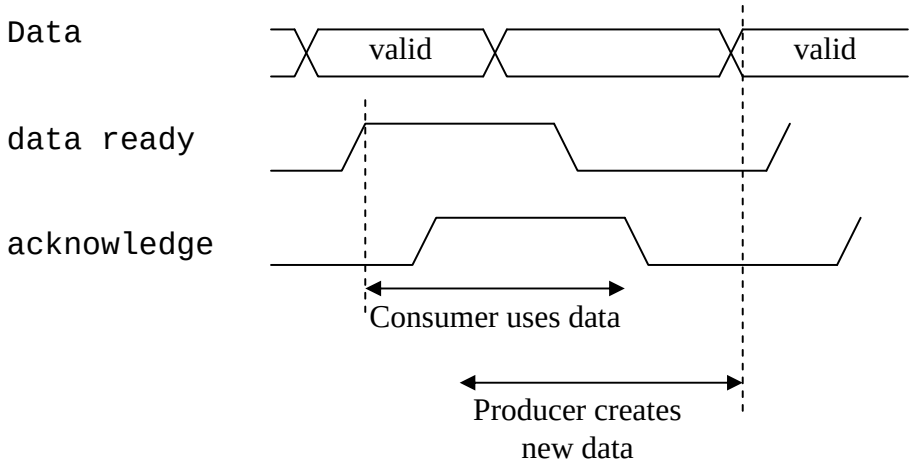

Figure 5: Two wire handshake

Note: A FIFO data-queue can create producer-consumer handshake signals automatically. The "data ready" signal is equivalent to the push into the tail of a queue, and the "acknowledge" signal is equivalent to the pop from the head of a queue. A flag bit, set by "data ready" and cleared by "acknowledge" coordinates the two tasks.

- **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 an output wire 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 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 6 shows the basic timing of the interrupt-driven handshake.

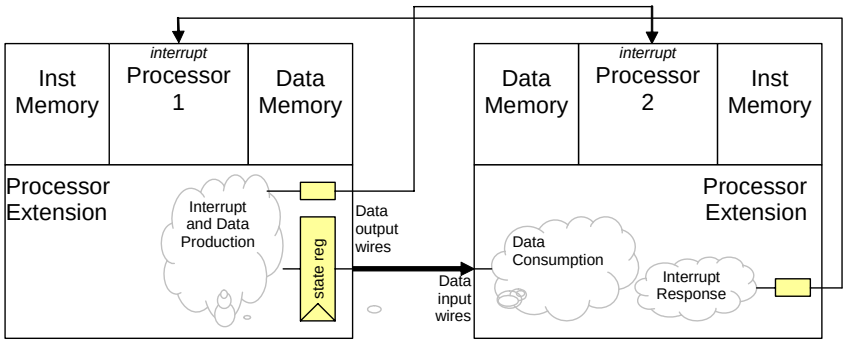

Figure 6: Interrupt-driven handshake

Data Queue Interfaces

The highest-bandwidth mechanism for task-to-task communication is hardware implementation of FIFO 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 transfer at a transfer rate of hundreds of MHz). This sort of interface is commonly used when designing hardware accelerator blocks but it's uncommon for processor-based communications because most processors cannot directly support FIFO queue interfaces. However, Tensilica DPUs support such interfaces and they are a valuable tool when a designer needs to boost processor I/O transfer rates.

For Tensilica DPUs, queue-interface widths need not be tied to a processor's bus width or general-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 FIFO 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 FIFO queue, assuming the queue is not full. If the FIFO 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 FIFO 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 interfaces 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 explicit 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.

Tailored, application-specific DPUs allow direct implementation of queue interfaces 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 uni-directional queue interface, like the one shown in Figure 7, allows data values of up to 1024 bits in width to be created and consumed independently of each other. A single customized DPU can support over 1000 interfaces.

A complex processor extension could 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 application-specific processors to be used for applications with very 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.

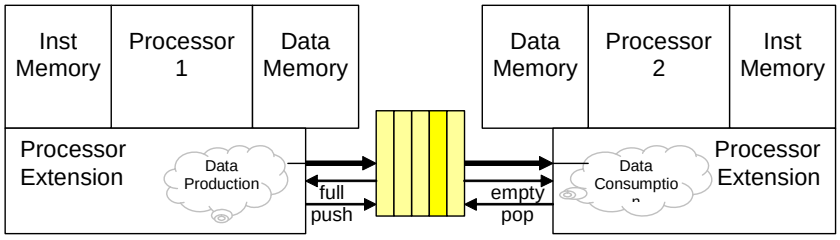

Figure 7: Hardware data queue mechanism

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 and ensure throughput at the average rate of producer and consumer, rather than at the minimum rate of the producer or the consumer. Sizing the queues is an important optimization and should be driven by good system-level simulation. If the queue is too shallow, the processor at one end of the communication channel may stall when the other processor slows for some reason. If the queue is too deep, the silicon cost will be excessive.

One processor can employ 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 destination-specific 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 would be combined into a 34-bit common queue, perhaps feeding a set of four 32-bit queues, as shown in Figure 8.

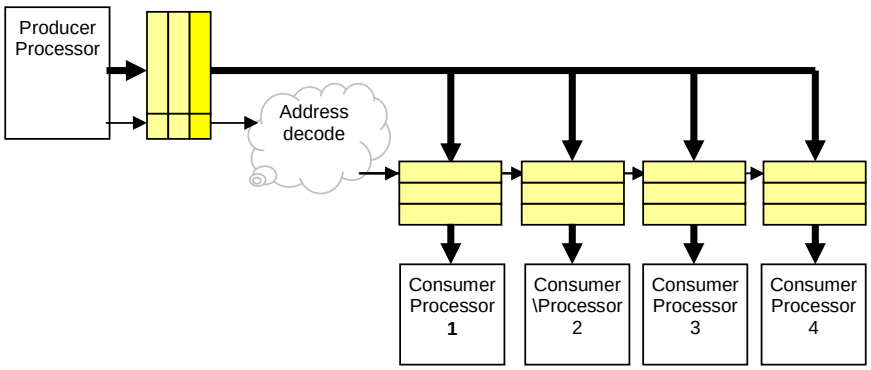

Figure 8: Producer encodes destination with data

Second, the FIFO 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 **9** shows a simple system with one producer and two consumers. The FIFO queues are mapped into the address spaces of the processors (here shown 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.

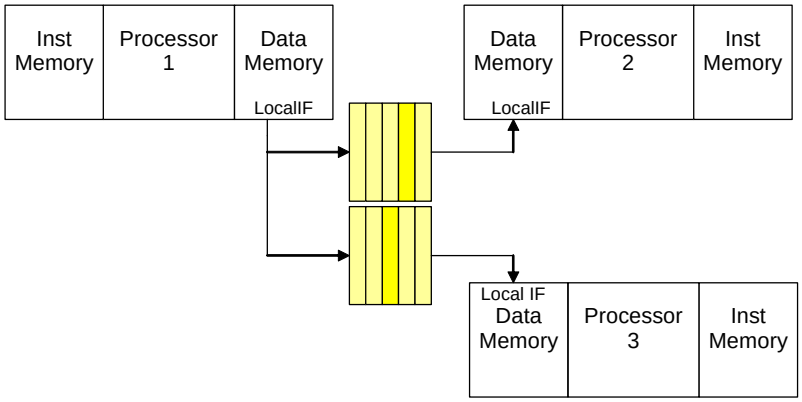

Figure 9: One producer serves two consumers through memory-mapped queues

The queue depth can be small if 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 10. When the two tasks pass data back and forth, the same register can be used for transfers in either direction.

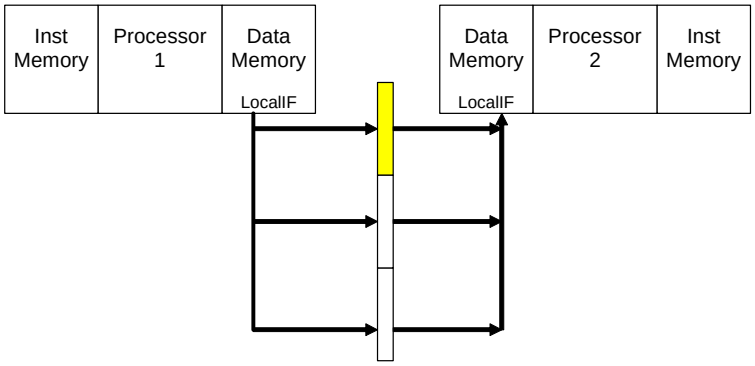

Figure 10: Memory-mapped mailbox registers

Memory-mapped and instruction-mapped FIFO 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 must be implemented in RAM or replaced with a shared-memory communication mechanism.

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 that work well between processors and between processors and other RTL blocks will help ASIC and SOC design teams create cost-effective designs in less time, with less effort, and with a lower risk of system failure.

Note: If you would like help in maximizing the on-chip communications methods used on for your next ASIC or SOC design, contact Tensilica for a consultation. See our locations at www.tensilica.com.