The Importance of Sockets in SOC Design

This paper introduces the *Open Core Protocol*™ (*OCP*™) within the context of contemporary System On a Chip (SOC) design issues. It explains why a standardized industry socket interface is essential to competitive SOC designs and it reveals how OCP fulfills the requirements for such an interface.

The discussion examines the necessity for SOC design acceleration to meet increasingly shorter time-to-market requirements as well as the advantages of subsequent reuse of developed Intellectual Property (IP).

Finally, the material reviews three specific implementation contexts that illustrate the flexibility that OCP provides semiconductor core designs.

The Problem

For years, relentless semiconductor fabrication advances and escalating market pressures have made time-to-market and design reuse a continuing topic within the semiconductor industry. Clearly, decreasing a SOC's development time can simultaneously decrease time-to-market. The design reuse practice is similarly simple and understandable – design once, reuse many, many times. But, decreasing SOC design times and achieving design reuse has proven elusive, indeed.

Every 18 months, these manufacturing improvements have historically increased circuit densities by a factor of two – called *Moore's Law*. This allows a given-size semiconductor die significant increases in scope and function at negligible, incremented manufacturing cost. For example, over the past five years, semiconductor gate complexity has surged from 200,000 to 500,000 gates to well over 10 Million, even 25 million gates. This is an increase of up to 50X and is the principal reason designers can produce SOCs.

Simultaneous with this increase in capability, designers have attempted to decrease the design cycle duration for both initial and derivative designs. This is in direct response to competitive market pressures that demand designs in as little as half the time, and which require frequent revisions because of continuously reduced product life cycles and feature enhancements. See Table 1.

	1997	1998	1999	2002	Delta
Process Technology	0.35m	0.25m	0.18m	0.13m	~7x
Gate Count	200-500K	1-2M	4-6M	10-25M	~50x
Design Cycle (months)	12-18	10-12	8-10	6-8	~2x
Derivative Design Cycle (Months)	6-8	4-6	2-4	2-3	~2x

Table 1. Increasing Complexity and Reduced Design Cycle Times
Source: *Surviving the SOC Revolution,* Chang et. al.

In summary, the math is very easy: designing semiconductors with fifty times the circuits in half the design time means 100 times the productivity, if you can achieve it. That's the

good news. The bad news is that this has proven to be an intractable goal. Worse, the proposition is becoming more difficult by the day, as complexity continues increasing and design cycles continue decreasing. The net effect is that time-to-market and core reuse are continuing casualties, along with product schedules and design efficiencies.

Shortening SOC Design Times

To address the time-to-market issue, first consider the benefits of designing individual SOC cores and the final SOC in parallel. Here, enterprises would clearly have a significant opportunity to reduce design time since all design aspects, including SOC simulations (timing and performance analysis, etc.) occur in parallel.

This reduces the SOC design time to that of the longest-effort, single-element design. The element might be an individual SOC core or, perhaps, the SOC integration effort. Either way, development schedule risk becomes bounded – assuring a higher probability of a satisfactory SOC within an accelerated development schedule. This also allows more predictable scheduling. Since all the development is bounded and all design is done in parallel, problems are not solved in serial fashion. This means problems are detected and solved sooner. The design flows become very predictable.

However, parallel development in this context mandates clearly defined divisions of responsibility for each core and shared SOC resources. That's because, cores would only perform their native functions without any system knowledge. For example, a PCI interface core or MPEG decompression core would perform native functions without having any specific knowledge of the SOC interconnect mechanism. Similarly, the interconnect mechanism would handle transport considerations such as arbitration, address mapping, and data movement, without knowledge of the functions any core provided. The good news here is that this methodology exists and has been well understood for years. Its name is *layering*.

Layering has been applied successfully to the network space to define responsibility levels at each layer. Each layer has its own functions and well-defined interface with other layers with which it interacts. The same is true for software. Each function or task has well defined functionality and interfaces. The layering approach has historically delivered excellent results in many different areas.

Layering to the Rescue

Layering naturally decouples system-processing elements from the system they reside in. The elements might be software modules within a larger software program, or, more importantly to SOC designers, semiconductor cores within a SOC. In either instance, the principles are usually the same. It also turns out layering benefits are often the same:

- Reduced design cycles
- Simpler Validation
- Increased IP reuse

Layering enables design teams to partition a design effort into numerous activities that can proceed concurrently because they are minimally dependent. Narrowing the scope of a single activity usually increases the odds that its implementation is correct and more easily verified. This can dramatically accelerate final product delivery schedules.

Layering also naturally enables core reuse in different systems. Here, in keeping with the layering scheme, other system resources handle remaining requirements without requiring understanding of individual core functionalities. Since an individual core's interface is independent of (decoupled from) the system, with the right core interface design, it can remain unchanged as the core is reused in subsequent system designs that support the interface. By selecting an industry standard interface, there is no added time for this reuse approach since all cores require such an interface. But, what then, would be the core interface? It turns out the answer to this question is a *socket*.

Sockets

For decades, Local Area Networks (LANs) grappled with issues that are now emerging for SOC designers. In the end, LAN designers created well-defined interfaces comprising both a physical connection and as well as protocols for exchanging information over those physical connections. The appearance of these industry conventions subsequently enabled the computing industry to provide independently developed and functionally diverse plug-and-play products that commercial enterprises assembled into highly custom LAN configurations. So, the method is both clear and proven.

Ideal SOC Socket Requirements

Ideally, a SOC socket would enable core designers to concentrate on their core functionality and the interconnects individually associated with them (e.g. SOC interconnect and USB, 802.11b, SDRAM, etc.). Similarly, SOC system integrators should be able to concentrate on SOC timing, core service bandwidth and latency requirements, and final floor-plan design independent of core functionality. The socket would therefore provide the necessary physical and exchange protocol delineation necessary to achieve this well-defined layering.

To achieve this, first note that an ideal SOC socket must necessarily be transport implementation (specific bus or other interconnect) agnostic. That is, SOC cores would interface to an inter-core transport mechanism via the interface, but the precise transport mechanics (computer-style bus, a cross bar, configurable on-chip network, etc.) would be unknown to the core.

This is essential, otherwise, core designs would instantiate transport knowledge within their designs, encumbering their reuse in SOC designs that used differing transport mechanics. A transport-unaware approach therefore, ensures implementation independence, also allowing the system designers to select the optimum interconnect for their system's needs.

Finally, because of the bandwidth requirement diversity, the ideal interface should allow designers to configure interface implementations along various dimensions. The dimensions include interface data widths required to meet bandwidth requirements, exchange handshake protocols, exchange acknowledgements, etc. This enables SOC designers to tailor core and SOC designs to minimize complexity and circuit areas while supporting core and SOC requirements.

- **OCP is a Core-Centric Protocol Interface:**
  - **Facilitates unrestricted delivery of *ALL* Core signals and features**
  - **Enables unconstrained interface bridge to ANY bus structure**

IP Core

OCP

Target Bus A

Target Bus B
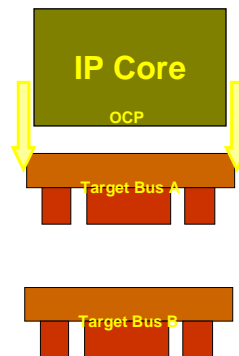
SOC Sockets

As we have now seen, the solution to maximizing core reuse potential requires adopting a well-conceived and specified ***core-centric*** protocol as the native core interface. By selecting an adopted industry standard, core designers not only enable core reuse for cores developed within their own enterprise, they also enable reuse outside their enterprise under Intellectual Property (IP) licensing agreements. Finally, they also maximize their ability to license and incorporate third-part IP within their own SOC designs. In other words, they achieve SOC design agility and the ability to generate revenue through IP licensing.

Moreover, a rigorous IP core interface specification, combined with an optimized system interconnect, allows core developers to focus on developing core functions. This eliminates the typical advance knowledge requirements regarding potential end-systems, which might utilize a core, as well as the other IP cores that might be present in the application(s). Cores simply need a useful interface that de-couples them from system requirements. The interface then assumes the attributes of a SOCKET – an attachment interface that is powerful, frugal, and well understood across the industry.

Via this methodology, system integrators realize the benefits of partitioning components through layered hardware – designers no longer have to contend with a myriad of diverse core protocols and inter-core delivery strategies. Using a standard IP core interface eliminates having to adapt each core during each SOC integration, allowing system integrators the otherwise unrealized luxury of focusing on SOC design issues. And, since the cores are truly decoupled from the on-chip interconnect, hence each other, it becomes trivial to exchange one core for another to meet evolving system and market requirements.

In summary, for true core reuse, cores must remain completely untouched as designers integrate them into any SOC. This only occurs when, say, a change in bus width, bus frequency, or bus electrical loading does not require core modification. In other words, a complete socket insulates cores from the vagaries of, and change to, the SOC interconnect mechanism. The existence of such a socket enables supporting tool and collateral development for protocol, checkers, models, test benches and test generators. This allows independent core development that delivers plug and play modularity without core interconnect rework. This also allows core development in parallel with a system design that saves precious design time.

Interface Solution Requirements

Core interface design requirements are very diverse and no single, specific implementation can possibly address them all. A standardized core interface specification needs to:

- Scale across a family of requirements
- Allow designers to configure specific interface instantiations along a number of dimensions (bus width, data handshaking, etc.)
- Address more than data-flow signaling
  - Errors
  - Interrupts
  - Flags and software flow control
  - Control and status
  - Test
- Capture *all* signaling between the core and the system

An OCP Introduction

OCP is a freely available, *bus-independent* protocol that meets all *core-centric* considerations discussed above. Specifically, it captures *all* of an IP core's communication requirements completely. As a highly-configurable interface, OCP is not a one-size-fits-all protocol. Rather, it comprises a continuum of protocols that share a common definition.

OCP explicitly supports sideband signals via optional extensions to the basic OCP data set. These sideband signals include, for example, reset, interrupt, error, control/status information, etc. In addition, a generic flag bus accommodates any unique core signaling

needs. An optional OCP test interface extension supports scan, JTAG, and clock control, enabling core debug and manufacturing test when integrated into SOCs.

System designers can therefore tailor a specific OCP configuration to match core requirements exactly. Through straightforward configuration procedures, OCP supports simple, low-performance cores with very simple and frugal OCP interfaces, while also supporting complex, high-performance cores with more complex interfaces.

An IP developer can therefore *complete* an IP core design using the OCP interface. No end-application knowledge is required beyond the OCP, allowing complete independence for members of the often global design teams. The system integrator is also free to choose the on-chip interconnect that best suits the system requirements of the application, then effectively "wraps" that interconnect to present OCP interfaces to the cores.
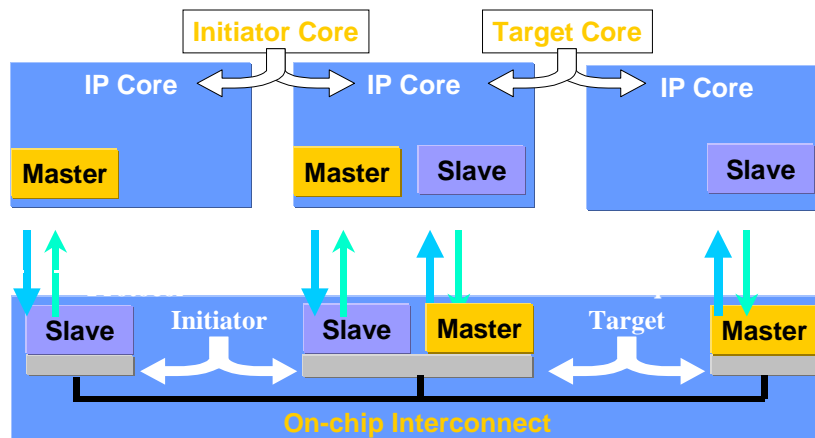
OCP-IP members receive the CoreCreator™ tool as an OCP protocol compliance environment and "packager" for all the representations necessary for efficient reuse of an IP core. It is available at no-charge to all OCP-IP members.

Example OCP Core Interfaces

These examples show how three very functionally different cores can use an OCP interface. The three examples are:

1. A bus bridge
2. A processor interface
3. A memory interface

The discussion first presents each core's OCP interface(s) in isolation. Then, it suggests additional common signals.

The signal names are part of the OCP protocol and the reader should reference the *Open Core Protocol Reference* manual for a full signal description. The OCP specification is freely available at http://www.ocpip.org.

In a nutshell, an OCP connection has a Master entity and a Slave entity. Some simple notes on OCP nomenclature and the protocol:

- The Master drives all signals having a name starting with the letter *M*
- The Slave drives all signals having a name starting with the letter *S*
- There are simple handshakes for the protocol
- The master and slave can both assert flow control
- All transfers and signals are synchronous to the rising edge of OCP clock

Example 1 - Bus Bridge

A bus bridge might interconnect a PCI, USB or other bus standard to OCP. The controller would have an external (to the SOC) PCI or USB interface and the internal SOC interface would be OCP.

Bus bridges usually act as a both a master and a slave on the internal SOC interconnect. The master sends the bus traffic to the desired location and the slave writes or reads the bus bridge internal control or status registers.

The slave consists of a simple OCP interface and most likely needs a few side band signals. A likely slave core OCP signal set for this example is:

- MCmd   Master Command (e.g. read/write)
- MAddr   Master Address (up to 32 bits)
- MData   Master Data (write data; 8,16,32,64,128 bits wide)
- SCmdAccept Slave Command Accept
- SResp   Slave Response
- SData   Slave Data (read data, must be the same size as MData)
- SError   Slave Error, error from the bridge
- SInterrupt Slave Interrupt, interrupt from the bridge
- Control   Control bits for the bus bridge
- Clk   The Clock signal
- Reset_N   The Reset signal

The interface uses that slave interrupt because this example's interface requires only one. If there were more than one interrupt, SFlags could provide up to 8 additional interrupts.

The bus bridge master might have the following signals:

- MCmd   Master Command

- MAddr   Master Address (up to 32 bits)
- MData   Master Data (write data; 8,16,32,64,128 bits wide)
- MBurst   Master Burst
- SCmdAccept   Slave Command Accept
- SResp   Slave Response
- SData   Slave Data (read data, must be same size as MData)
- Clk   The Clock signal
- Reset_N   The Reset signal

With PCI, optional OCP thread signals might enhance the interface. This would allow the interface to support concurrency and out-of-order processing of transfers. The optional OCP *complex extension* signals support multiple threads. Transactions within different threads have no ordering requirements, and can be processed out of order. Within a single thread of data flow, all OCP transfers must remain ordered. Threads for a PCI interface might be useful to access different memory and I/O regions.

- MThreadID   Master Thread Identifier (up to 16 different threads)
- SThreadID   Slave Thread Identifier (up to 16 different threads)

Example 2 - Processor Interface

A processor interface usually only requires an OCP master. The signals would be similar to the bus bridge's master but would normally include byte enable signals for less than single word transfers.

A likely core OCP signal set for this example is:

- MCmd   Master Command
- MAddr   Master Address (up to 32 bits)
- MData   Master Data (write data; 8,16,32,64,128 bits wide)
- MBurst   Master Burst
- MByteEn   Master Byte Enable
- SCmdAccept   Slave Command Accept
- SResp   Slave Response
- SData   Slave Data (read data, must be the same size as MData)
- SError   Slave Error, input to the processor
- SInterupt   Slave Interrupt, usually the NMI pin
- SFlag   Slave Flags, other interrupts to the processor (up to 8 flags)
- Clk   The Clock signal
- Reset_N   The Reset signal

Some newly available processors support concurrent instruction and data cache-miss fetches. OCP threads directly support this. Hence, the following signals could be needed to enhance concurrent memory operations for such processors.

- MThreadID   Master Thread Identifier (up to 16 different threads)
- SThreadID   Slave Thread Identifier (up to 16 different threads)

- MThreadBusy   Master Thread Busy
- SThreadBusy   Slave Thread Busy

The master and slave thread busy signals permit each thread flow control. Processors might have several fetches outstanding, but not the resources to handle all of them simultaneously. The thread busy signals therefore enable an OCP master processor or the target slave to control the transfer flows as necessary.

Example 3 - Memory Subsystem

A memory subsystem may interface to DRAM, DDR, SRAM or FLASH. The OCP signals require some OCP *complex extensions* to maintain high bandwidth utilization. The memory subsystem is probably multi threaded, to service multiple memory banks. The memory controller will also have OCP *simple extensions* such as *burst* and *byte enable* to service requests efficiently.

A likely core OCP signal set for this example is:

- MCmd   Master Command (e.g. read/write)
- MAddr   Master Address (up to 32 bits)
- MData   Master Data (write data) (8,16,32,64, or128 bits wide)
- MBurst   Master Burst
- MByteEn   Master Byte Enable
- SCmdAccept   Slave Command Accept
- SResp   Slave Response
- SData   Slave Data (read data, must be the same size as MData)
- MThreadID   Master Thread Identifier (up to 16 different threads)
- SThreadID   Slave Thread Identifier (up to 16 different threads)
- MThreadBusy   Master Thread Busy
- SThreadBusy   Slave Thread Busy
- Clk   The Clock signal
- Reset_N   The Reset signal

The memory subsystem might also utilize a larger bit width on MData and SData than the memory to which it interfaces. This makes each system interconnect transfer maximally efficient.

Signals Common to Each Example

Each of the preceding examples can have scan and JTAG signals. These test and scan signals might be common to each core, but the number of scan chains might differ. OCP allows test structures as part of the interface, not as a separate entity. This completes the socket versus merely addressing data flow considerations.

Having scan and JTAG access to each of the example cores might require the following signals:

- ScanCtl   Scan Control
- ScanIn   Scan In (up to 256 scan-in ports or chains)
- ScanOut   Scan Out (up to 256 scan-out ports or chains)
- ClkByp   Clock Bypass, use Test clock instead of normal clock
- TestClk   Test Clock
- TCK   JTAG Test clock, uses IEEE 1149.1 definitions for all signals
- TDI   JTAG Test In
- TDO   JTAG Test Out
- TMS   JTAG Test Mode Select
- TRST_N   JTAG Reset

Conclusion

A standard socket core protocol is essential to the SOC design community. OCP is the ONLY complete, fully supported, and proven socket. Immediately adopting OCP avoids incompatible or proprietary solution proliferation and expands the total available market for commercial and legacy IP cores.

The complete, fully supported core-centric OCP delivers substantial and demonstrable benefits over older style bus-centric protocols. OCP is a core-centric, openly licensed, royalty-free core interface protocol. It does not restrict or otherwise interfere with inherent core capabilities. It is scalable and configurable to match different communication requirements associated with different core and SOC designs.

Cores with OCP interfaces and OCP interconnect systems enable true modular, plug-and-play integration, allowing the system integrators to choose cores optimally and the best application interconnect system. This allows the designer of the cores and the system to work in parallel and shorten design times. In addition, not having system logic in the cores allows the cores to be reused with no additional time for the core to be re-created.

Finally, verification and test suites, when written to OCP specifications, are completely portable across multiple designs, infrequently requiring even minor adjustments for a particular interface bridge.

The OCP specification is freely available at http://www.ocpip.org.