

Chapter 14

SOCKET-BASED DESIGN USING DECOUPLED INTERCONNECTS

Drew Wingard

Sonics, Inc., Mountain View, CA, USA

wingard@sonicsinc.com

Abstract: Advances in deep sub-micron semiconductor process technologies offer the SoC designer the promise of more functionality than can be realized using existing tightly coupled architectures and EDA tools. As the ability to integrate IP cores increases linearly, the complexity of interconnecting these cores also increases, often geometrically. The overhead in design, integration, and verification of inter-core communications quickly becomes greater than the design savings in using pre-verified functional cores. The solution requires a matching of interconnect implementations to computational blocks. Socket-based design is a design methodology that can greatly reduce the time and effort expended on design and verification of complex SoCs. We will discuss the challenges of tightly coupled design, explain why decoupled interconnect design is essential, define socket-based design; explore the OCP socket specification and give examples of both processor-centric and I/O-centric SoC design using OCP-based sockets and decoupled interconnect cores.

Key words: socket-based design, IP reuse, smart interconnect IP, OCP socket standards, SoC design methodology, interconnect cores

1. INTRODUCTION

Today SoC (System-on-Chip) devices of greater than 100 million-gate complexity can be developed using 90-nanometer semiconductor process technologies. But existing design architectures, EDA tools and IP core-based design techniques limit the design potential of SoCs to much lower complexity levels, particularly for manageable design teams and timelines. Designing multifunction, system-level SoCs at this complexity level requires

new design approaches and a radical shift in thinking. The chip design and EDA industry has historically focused on the development of functional blocks rather than the development of interconnects. But today, a shift must occur as interconnects become the gating factor in developing successful SoC devices. Interconnect architectures must now match the sophistication of functional core architectures if the industry is to realize the potential of deep submicron silicon process technology, a point noted by Mead and Conway in 1980 [1].

Historically, electronic system design has focused on functional design. When systems consisted of processor, memory and I/O devices on printed circuit boards, the interconnect was the passive connection between active chip-level functional blocks. Abstractions such as buses and control signals were used to group chip-to-chip communications into logical clusters. With improved process technology, integration of these devices became feasible and design methodology maintained the passive interconnect model from board-level design. At the level of one or two processing cores surrounded by a handful of I/O blocks and a single memory controller, design tools and designers are capable of managing the complexity of the total system on chip. However, as the integration of many heterogeneous processing units with tens to hundreds of I/O blocks becomes possible, the task of managing the entire design and the complexity of the interconnect schemes are overwhelming the design team and their traditional EDA tools. The focus of much of the industry remains only on the functional aspect of design. As this chapter will explain, the interconnect side of design must now evolve from passive and tightly coupled to active and decoupled in order to support the integration of large numbers of functional cores in a SoC device.

2. EVOLUTION OF DESIGN ABSTRACTIONS

Traditional approaches to managing system design complexity rely upon the creation of abstractions. Abstractions serve to both minimize the number of elements that are managed directly and to simplify the interactions between the elements. The refinement of electronic system design approaches that has led to SoC design has made significant use of design abstractions, particularly with respect to functional design. As shown in Figure 1, the level of functional abstraction has progressed from gate to block to core, and will be migrating to tile-based abstractions.

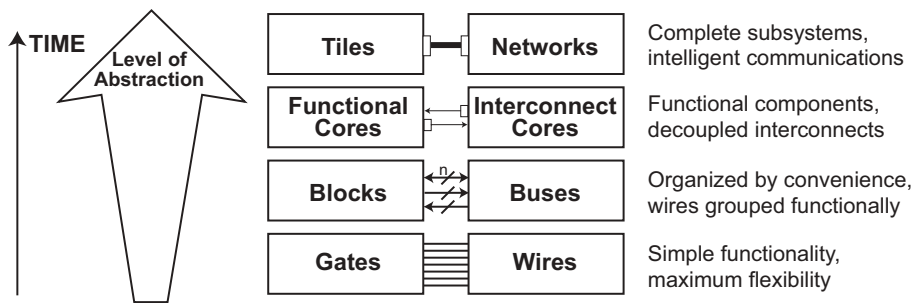


Figure 14-1. Evolving levels of design abstraction

Gates present the simplest functionality and have the least limitations in how wires are used to connect them. Tiles present the highest functionality and leverage a high-level network protocol to efficiently structure and optimize inter-core communications.

In between, blocks are designed using HDL rather than gates and have boundaries based on function and convenience that enable the placement of multiple, identical copies throughout a design, simplifying design and verification. Buses are used to group wires with similar functions together and provide some abstraction of communication through vectors of signals and simple protocols. Most interfaces between blocks have been evolved in ad-hoc fashion rather than having been designed for optimum system integration. Since the functional blocks are the key design element, communication between them appears free. At this level, integration is tightly coupled by definition and block functionality drives the design.

At the next level of abstraction, we would expect a reasonably sized functional core to solve a significant portion of the design problem transparently from the rest of the system. With multiple design teams developing cores and integrating them together, the design methodology must support independent and concurrent design. This abstraction must therefore embrace both intelligent functional partitioning and communications abstraction. Embedding communications logic into these functional cores defeats this purpose. The communications logic should be accumulated into a separate interconnect core. This active interconnect core manages the communication between the functional cores to meet the system level requirements. The result is that each functional core is decoupled from the system-level communications responsibilities.

The tile and network abstraction is above that of functional and interconnect cores. Tiles are defined [3] as complete functional subsystems that can be connected using on-chip networks. Such a tile will execute a set of functions with little or no dependence upon the rest of the device. These self-sufficient operating characteristics may typically be implemented in an

embedded processor in the tile. A tile normally manages its own I/O locally. It communicates with other tiles via messages composed of data structures rather than through transactions composed of data words. Since tiles normally contain sufficient local memory for latency-critical traffic, on-chip networks offer an attractive interconnect for routing tile-to-tile messages. Tiles are either application specific, such as a GSM modem or a GPS receiver, or generic, such as a host processor for application software or a reconfigurable computing fabric. An example of a generic tile without a processor would be a shared memory subsystem that provides storage services to other tiles.

Elements at each level of abstraction are composed of elements from the lower abstraction. For instance, an interconnect core is composed of blocks and buses. The result of each abstraction is reduction in complexity by minimizing the number of objects, and their interdependencies.

2.1 Functional Cores and Buses – Mismatched Abstractions

Today, it is typical for the SoC architect to begin by defining the system-level functionality, partitioning the design according to known guidelines that define roughly what can be implemented in a single-chip design (area, power consumption, timing characteristics), and then passing the design on to the functional core designer, the interconnect specialist, and the physical layer designers to implement the final silicon device. Ideally, this approach should enable the rapid development of complex ICs using pre-verified functional cores and tiles. As the number of pre-verified cores increases over time, the process of developing systems should become simpler and faster by reusing these proven functional cores. Ideally, this should lead to “SoC design by feature combination.” [2]

The reality is that as more functional cores are added to a design, the system completion time increases dramatically. The flaw lies in the increasing complexity of the inter-block interconnections and the lack of system-level modeling and design tools. It has been estimated that 30 to 70 percent of the design effort of a complex SoC design is now in designing and verifying the interconnect [4].

A useful way of understanding how this occurs is to examine the underlying abstractions used to implement an SoC and how they are mismatched in a typical design.

The reason to move to a higher level of abstraction is to isolate the designer from underlying implementation complexity so each abstracted element can be used independently. Thus a designer integrating functional cores should be isolated from the intricacies of blocks and buses. However,

today's SoC designers typically attempt to integrate functional cores using passive buses. This requires the core designer to embed awareness of low level communications schemes within the functional core. Such an approach forces all functional cores to be "bus aware." Embedding communications logic into the functional core results in a fragile design element that is not independent of the intricacies of other cores or the system-level communications scheme. The embedded low level bus functionality forces the functional core to interact at its level, essentially turning the core into a highly complex, but still low level block element. This complexity is evident to the SoC integrator, who must then work at too low a level. The functional core abstraction has failed.

Because of this mismatch, there are numerous problems that occur at the core integration stage as shown in Figure 2. At each stage of integration and verification, designers discover system-level interactions that cause reengineering of the functional cores. At first integration, signaling and protocol mismatches are frequently detected due to poor specification of inter-core communications. When revalidating a core's functionality in the context of the integrated SoC, functional defects are discovered as transaction ordering and timing is discovered to be different than what was anticipated. The logic synthesis and timing analysis processes often uncover unexpected timing arcs that cross functional core boundaries. Later the placement and routing process, where the actual wire lengths between the functional cores are first discovered, uncovers further timing challenges. Finally, if the schedule permits, system-level performance verification, which attempts to verify that the intended application will perform properly on the SoC, is likely to uncover new problems. Each time these functional cores are re-engineered, their verification test benches, synthesis scripts, and test plans must be updated to match the revised core definition.

To achieve the benefits of concurrent SoC design, functional cores must become decoupled from each other. This will avoid the necessity of redesigning functional cores as the system-level communications are refined into the interconnect core.

Because of the problems of long design and verification cycles, the current SoC design approaches are no longer able to fully take advantage of the available on-chip gates. Some observers point to a current lack of front-end system-level design tools and the inability of existing front-end design tools to account for back-end physical layer realities at the beginning of the design as the cause of a growing design productivity gap [5]. This gap is defined as the difference between available on-chip gates and the actual gates used. Better tools may improve, but not solve this problem.

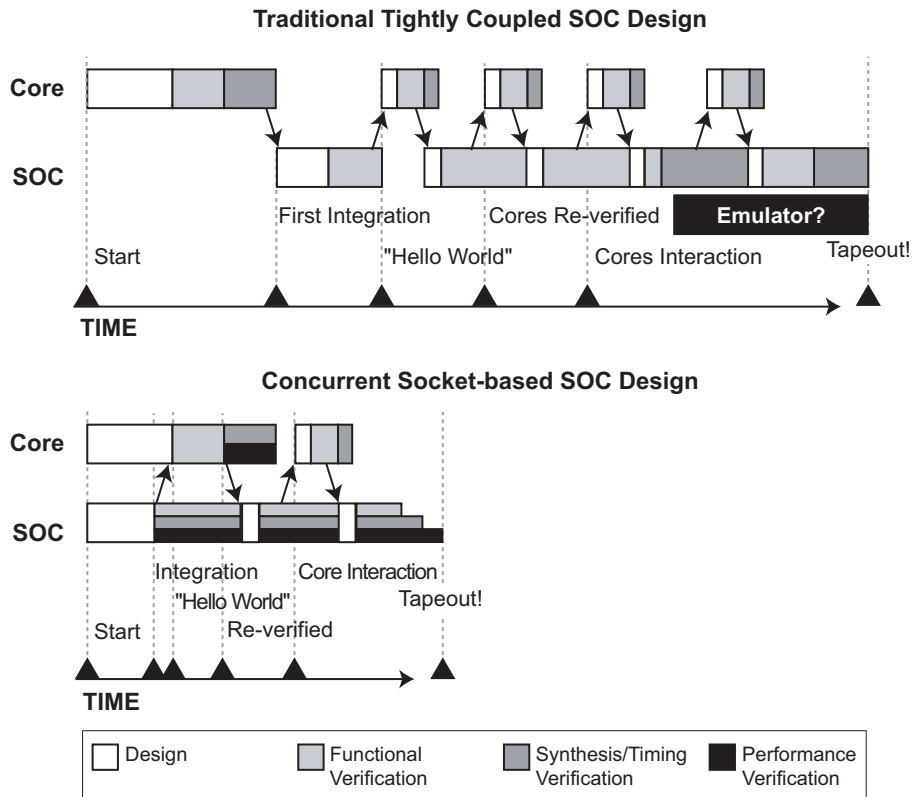


Figure 14-2. Comparison between tightly coupled and concurrent socket-based SoC design

2.2 Leveraging Functional Core and Interconnect Core Abstractions

The problems of long development times and the inability to realize on-chip gate potential can be resolved by eliminating the mismatch of functional cores with buses.

The most obvious solution is decoupling the computational tasks of the functional cores from inter-core communication tasks. The thesis of this chapter is that the practical approach to decoupling is to isolate the functional cores using a standard interface socket and to optimize inter-core communications using an active interconnect core. By applying these decoupled design techniques, a concurrent design methodology can be achieved using existing design tools and styles.

A concurrent design methodology can be implemented in the following manner. First, the design team can begin to separate the functional core from

the intricacies of the inter-core communications task by adopting a standard functional core interface socket. Second, by shifting from a “passive wire” model of inter-core communications to an “active interconnect core,” the designer can develop new interconnect cores that are independent of core functionality and that can be independently optimized for both local and system-wide communications. This decoupled approach will speed the development of complex SoCs, yield more highly optimized functional and interconnect cores, and greatly increase the potential for core reusability. This approach enables the benefits of NoC (Network On Chip) based platforms using the “design by feature combination” technique put forth by Jantsch and Tenhunen in 2003 [2].

A potential candidate for a standard functional core interface is the OCP (Open Core Protocol) interconnect specification. This flexible, scalable and well-defined socket is processor-agnostic and capable of supporting a wide range of interconnect schemes. Most importantly, it is enjoying growing and widespread support throughout the industry [6].

The shift in modeling from the passive wire to the active interconnect core has already begun as designers are realizing that interconnect cores can be the architectural underpinnings of modern multifunction and reusable SoC designs [7]. A decoupled interconnect-based architecture can deliver scalable latency and bandwidth where needed while enabling separate optimization of its component functional and interconnect cores. The rest of this chapter outlines the challenges of traditional tightly coupled design, describes the requirements for a standard socket, defines the goals of a decoupled interconnect core, uses the OCP socket standard as illustration of how functional cores can efficiently communicate, and presents two decoupled interconnect options as solutions to application specific requirements found in complex SoCs.

3. LIMITS OF TIGHTLY COUPLED DESIGN

At first glance, tightly-coupled design, where IP blocks or functional cores are tightly coupled to each other, would seem to be the most efficient solution to most design problems. One would expect the smallest silicon area and the highest performance when active IP cores are connected to their associated subsystems with the simplest of passive connections (wires) and no additional overhead. Certainly, this system-level interconnection scheme will offer maximum flexibility, as there are no limits on how the interconnection between subsystems takes place.

But these advantages come with some important limitations even in the simplest systems. The expertise gained in accomplishing a given

interconnect task, e.g. interfacing one processor to a given bus structure, can be best leveraged in future designs only through shared experience between members of the design team. Put more generally, design team members must work closely together in order to achieve optimum results.

The essential fact limiting tightly couple design is that communication is no longer free – wires are no longer elastic, zero delay elements connecting cores or blocks. The increasing relative wire delays at the distances associated with connecting far-flung cores across the chip plays a major role in the performance of the system. The implication of wire delays approaching or passing cycle times is that the transformation of the functional block diagram into the physical floor plan has profound performance implications. Managing this transformation requires active interconnect cores that isolate functional cores from inter-core communication issues and enable accurate up-front modeling of communications.

In addition to the difficulties in resolving low level issues such as signal timing issues at inter-core boundaries, there are the challenges of managing a large number of inter-dependencies as the design grows in size and scale. Inter-dependencies are much harder to manage when design teams grow to multiple groups in multiple locations, and changes are more difficult to implement. Worse still, it becomes increasingly difficult to understand the complete set of inter-dependency constraints. Whenever one core is inter-dependent upon another, the dependency must be managed at the system level. Design complexity using tightly-coupled design techniques overwhelm the designer's mental capacity to understand the total design interactions.

As tightly coupled designs scale upward, it becomes increasingly difficult to create levels of abstraction that promote understanding or enable reuse. This leads to longer design times as poor models of the electrical, logical and protocol behaviors lead to verification issues that require many design iterations. At the SoC design level, there is no formalism, and thus no tools, available to address all of the interactions among tightly coupled cores. Design teams must deploy a better, more decoupled architecture.

As a result, in actual practice, tightly coupled design of systems deploying large numbers of heterogeneous processors is often the least efficient solution in terms of silicon efficiency, design time and cost, flexibility and re-use of IP. Because of the complex interactions and inter-dependencies of the electrical, logical and protocol components, design iterations intended to result in successive design refinement, actually result in successive system and core redesign.

4. TIGHTLY COUPLED VS. DECOUPLED DESIGN

We have seen that tightly-coupled designs are vulnerable to significant redesign due to electrical, logical, and protocol problems detected at each phase of integration and implementation. With such an approach, the redesign occurs inside the functional units, making IP core re-use highly unlikely. This section provides a comparison between tightly-coupled and decoupled design approaches.

Processor-centric on-chip bus protocols such as AMBA (ARM) [8] and CoreConnect (IBM) [9] are modeled on architectures designed for traditional printed circuit board-based systems. They feature predominately single processor-to-memory and processor-to-input/output device transactions. These blocking bus architectures do not support simultaneous, heterogeneous data traffic among multiple processing elements such as those found in modern SoCs. In particular, such architectures do not easily support isochronous data flows often found in networking, multimedia and digital consumer applications. In such applications it is common to find several processing elements (primary system processor, DSP unit, MPEG encoder/decoder, and multiple communication processors) accessing shared memory and I/O resources throughout the chip.

When developing a SoC using bus-centric interconnects, the resulting system is a mixture of processor subsystems linked together using computer buses and a collection of point-to-point links dependent upon the functionality of the on-chip elements as shown in Figure 3. The trouble with such schemes is that they exhibit behaviors (power consumption, signal delay and distortion, cross-talk and noise) that are extremely difficult to predict or model before the final physical layout stage. Since interconnects can be responsible for up to 90 percent of the global signal delay in a submicron design [10], this means that meaningful timing verification cannot begin until late in the design cycle.

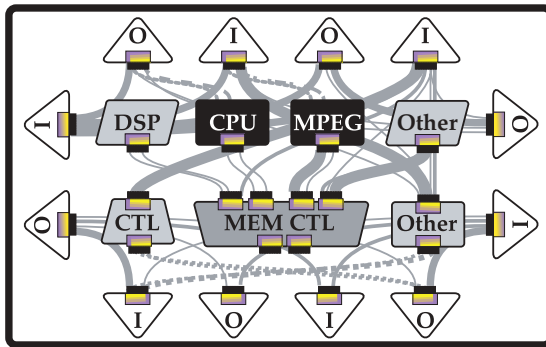


Figure 14-3. Passive interconnect scheme using multiple buses and point-to-point links

At the other end of the design spectrum, a socket-based design approach using active interconnect cores isolates the functional core from the details of the on-chip interconnect architecture as shown in Figure 4. This reduces design time by freeing the core developer to focus on core-related issues rather than system-level interconnect issues and by providing interconnect characterization that can be used to accurately model the entire SoC long before final physical layout begins. In addition, socket-based design results in highly optimized functional cores and tiles that can be easily reused in other systems with no additional redesign.

In the figure, the interconnect cores are represented as multi-drop connections between functional cores. Internal interconnect core topology is unspecified. The core sockets form the connection between the functional and interconnect cores. Tiles are represented as encircled collections of functional and interconnect cores.

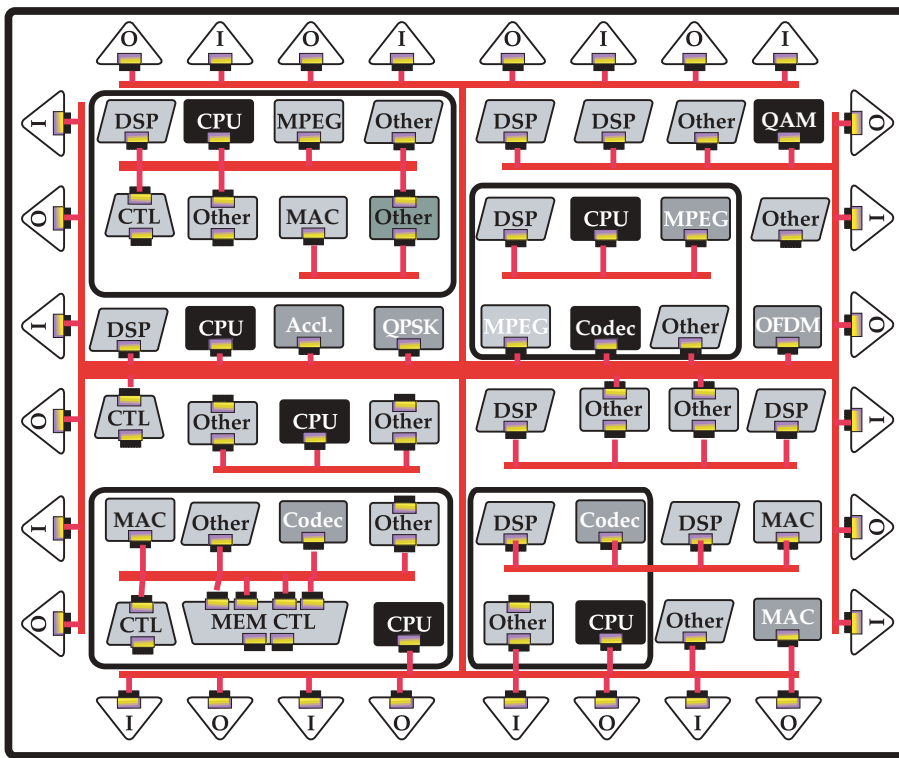


Figure 14-4. Socket-based design concept using decoupled active interconnect cores

Decoupled design methodologies are further needed to respond to the nature of today's electronics business environment. As globalization of the electronics industry has taken hold, there are greater requirements for

dispersed design teams working independently. Functional core development must proceed independently from specific device implementation, since the device targets can change during design and the core may be needed for reuse in future designs. Core designers must anticipate the integration and design verification stages to avoid core redesign. Standard interconnect techniques must be deployed that separate core computational functions from inter-core communications and a standard core-to-system interface must be supported.

4.1 Decoupled Interconnect Cores

Only decoupled interconnect cores are able to meet these design constraints. One of the key characteristics of the decoupled interconnect core model is that it enables the selection of independent clock frequencies, data widths and transfer protocols to satisfy both functional core and system-level requirements. To do this, it must be capable of managing total system communications by supplying the required throughput at acceptable latencies.

A decoupled interconnect core can provide an optimum local operating environment for each functional core that meets the core's performance constraints. The functional core designer chooses the appropriate functional and communication architectures for their core, including data path widths, degree of transaction pipelining, and internal buffering. In addition, the designer specifies required bandwidth and latency constraints that the core imposes upon the system as a function of the core's operating mode and/or clock frequency.

While each functional core can now be optimized for its specific functionality, the decoupled interconnect core can be similarly optimized to improve system-level communications between all cores. The interconnect core manages the inter-core communications, satisfying functional core communications requirements while optimizing system performance, power, and area to meet SoC requirements.

System level management includes establishing routing between functional cores, providing an access control mechanism that meet QoS standards in terms of latency and throughput, and generating, routing and monitoring sideband signals. At the silicon floor planning and physical implementation stage, the decoupled interconnect core must be able to span the distances between functional processor core and I/O core placements and to optimized the frequency/latency trade-offs to fit application requirements. It should deliver robust system services such as error management, security/protection, identification, power management and event

management, while presenting a set of predictable physical design characteristics that can be easily integrated into existing design tool flows.

In the design examples, we will illustrate two decoupled interconnect IP models that address these topics for processor-intensive applications and for I/O-intensive designs.

5. SOCKET-BASED DESIGN

Socket-based design is a method of using a pre-specified, standard interface to provide electrical, logical and functional connections between cores. Note that the socket need not care about the differences between functional and interconnect cores. The socket is equally applicable for connections between functional core to functional core, functional core to interconnect core and interconnect core to interconnect core. The socket defines the boundary of responsibility between the cores allowing them to be independently designed and verified, with the assurance that they will function correctly when connected.

A complete socket model should provide the designer with a range of abstractions to support various electrical, logical and functional behaviors. At the electrical level, the socket is implemented using wires. At the logical level, the socket uses protocols. At the functional level, the socket delivers transactions. Electrical measures include delay in seconds, capacitive loading in Farads and activity rates in cycles per second. Logical measures include delay in cycles of latency, and activity in peak bandwidth capacity. Functional measures include latency constraints and throughput requirements.

For a socket model to succeed it must have support for and within the existing industry infrastructure. This means that bus functional models, protocol checkers, performance analysis tools, formal property checkers, and multiple language support (HDL, verification language, C++) should be readily available.

To simplify implementation, a socket should be a point-to-point link using only unidirectional, synchronous signaling. This interface model will support the easiest design flow integration, a key consideration of any proposed standard.

To be effective, socket-based design should support the needs of the functional core. This means that communications protocols and timing characteristics must be tuned to optimize for the natural behavior of the functional core. The incorporation of a socket mechanism enables a natural isolation of the functional core and relieves the core from unnatural restrictions in electrical, logical, or functional behavior based on inter-core

communications parameters. To enable this isolation, the socket itself must be capable of supporting a wide range of signaling and transfer protocol parameters and be configurable and extensible. Parameterized aspects could include selecting an address width, a data width and the number of interrupts, etc. Configurability could include the ability to enable or disable specific protocol capabilities, tune pipelining, select handshaking/flow control, support threading, select sideband signaling, add command extensions and select burst options. Extensibility could include adding core-specific functionality such as supporting security modes and adding parity or ECC features to transactions.

Socket-based design techniques add value to the SoC design process by allowing the designer to simplify communication issues to local point-to-point transactions at core boundaries, using protocols natural to the core. This approach simplifies the initial design of a core, eliminates artificial performance loss due to bus-centric interface choices, and enables greater independence of the design process. This greater independence facilitates both independent creation and verification of cores as well as much higher reuse of existing core designs. In addition, the socket enables the creation of a wide variety of decoupled interconnect cores using advanced internal protocols that must simply maintain the socket boundary.

6. THE OCP SOCKET

The OCP socket standard simplifies the interconnection of functional and interconnect cores and promotes core reuse. It is an excellent example of an open standard socket in use today. In brief, it is parameterized, configurable and extensible and is well supported by design and verification tools.

Initial efforts to define a standard core interface socket were begun in the mid-90s with the formation of the VSI Alliance (VSIA) [11]. VSIA's ambitious goal was to establish standards that enabled the integration of IP blocks from multiple sources. VSIA defined the Virtual Component Interface (VCI) as its IP core interface. While a valuable first step, this effort did not result in a widely adopted standard.

The non-profit Open Core Protocol International Partnership (OCP-IP) was formed in 2001 to support the first open and complete IP core socket for plug-and-play SoC design. The Open Core Protocol socket is a complete specification capable of supporting a wide range of core communication requirements, bus structures and interconnect technologies.

The OCP socket is bus-independent and handles the three dominant communication types between cores on an SoC – data, control and test. It is supported by a complete set of design, validation, and implementation

support tools as well as a SystemC transaction interface and technical support documentation.

The OCP socket is a configurable interface that supports a basic set of data flow signals and an extended set of sideband signals for control and status information. A generic flag bus supports specialized inter-core signaling and a test interface support scan, JTAG and clock controls to simplify SoC debug and test.

The OCP socket can be configured to meet specific core and interconnect requirements. One configuration of OCP can form a simple, low-performance core interface. Another configuration can offer more complex, higher-performance interfacing for advanced processors and memories. The SoC architect can implement whatever system-level interconnect that meets the needs of the SoC, while individual core designers can independently deliver their functional core using OCP.

As defined by the Open Core Protocol Specification [12], an OCP socket uses a master/slave framework with synchronous unidirectional signals sampled by the rising edge of the OCP clock. The OCP socket is fully synchronous with no multi-cycle timing paths and all signals (other than clock and reset) are point-to-point. These constraints simplify timing analysis and physical design.

OCP accomplishes data flow by explicitly separating transfers into a set of phases. In the request phase, the Master provides the Slave with enough information to sequence the transfer. For simple writes, the request phase will accomplish the data transfer. In the response phase, the Slave provides the Master with indication about transfer completion. For reads, the response phase will include the data transfer. These phases may be pipelined with respect to each other, so the Master may attempt to sequence multiple requests before the Slave has responded. A rich set of handshaking and other flow control signals allows each side to pace the transfers and the pipeline depth.

The basic OCP signal set as shown in Figure 5 includes Master command, address and write data signals (MCmd, MAddr, and MData), Slave handshake, response, and read data signals (SCmdAccept, SResp, and SData), the clock (Clk) signal and the reset (Reset_N) signal. This configuration supports basic address-mapped data flow communication between cores.

If the core needs more communication flexibility, additional OCP signals can be configured for the socket. These include burst controls and sideband flag signals, which are especially useful for supporting interrupts.

To support concurrency and out-of-order processing between transactions, OCP supports the concept of independent threads sharing a single socket. Transactions issuing on different threads have no ordering

requirements, and so can be processed out of order. Within a single thread, all OCP transactions must remain ordered. OCP threads are similar in many ways to *virtual channels* [13], with slightly more restrictive end-to-end mapping semantics.

Threads are added by configuring Master and Slave thread identifiers (MThreadID and SThreadID) and optional thread busy (MThreadBusy and SThreadBusy) signals for the socket. The thread busy signals allow the receiving device to assert per-thread flow control, which allows cooperating Masters and Slaves to implement truly non-blocking semantics across an OCP socket. This approach prevents a stalled thread from impacting the progress of other threads, eliminating deadlock scenarios while improving OCP's ability to support QoS guarantees.

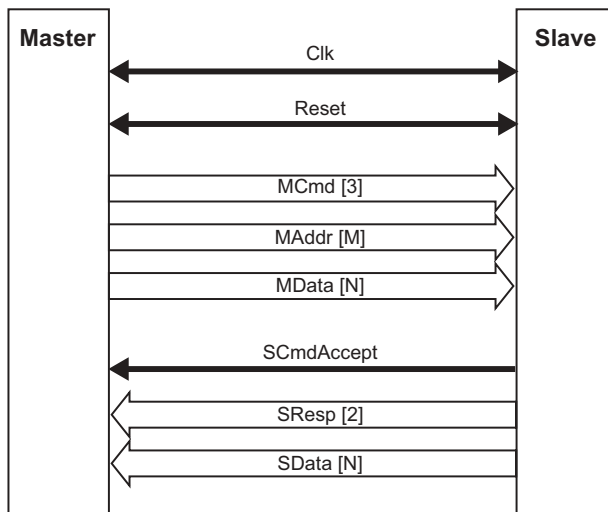


Figure 14-5. Basic dataflow signals of the OCP socket

A core developer selects a configuration of OCP to match the communications needs of the core, thereby creating a unique socket. For example, a UART designer might choose an OCP socket featuring an 8-bit data word (MData and SData), 3 address bits (MAddr), simple handshaking (via SCmdAccept), and a single interrupt signal (SInterrupt).

In contrast, the designer of an SDRAM controller might choose an OCP socket featuring 128-bit data, 28-bit address, deeply pipelined reads and writes, and 8 independent threads to support effective utilization of the multiple SDRAM banks. By implementing the full set of handshaking and flow control options, the designer can guarantee that the threads will stay independent, rather than blocking each other in either the request or response pipelines.

The Open Core Protocol was defined to be an excellent core interface socket. SoCs designed using OCP-compliant functional and interconnect cores realize the promise of socket-based design.

7. USING DECOUPLED ACTIVE INTERCONNECTS

As noted earlier, the adoption of a standard functional core interface standard such as the decoupled OCP socket is part of the concurrent SoC development solution. The second part is the deployment of interconnect architectures based on active interconnect cores.

The recognition of the need to decouple the computing tasks of the functional core from the inter-core communication tasks of the system is the basis of “communications-based design” [14]. A decoupled active interconnect methodology can be the foundation for this approach.

Decoupling frees the active interconnect core to be independently optimized for the targeted SoC, resulting in better overall results in much less time. The resulting SoC designs are more scalable. The decoupled active interconnect core supports increasing logical connectivity and increasing total communications bandwidth by increasing the total amount of communication resource, without requiring changes to any of the functional cores.

From the perspective of the active interconnect core, the functional cores appear to be simply communications sources and sinks. Each functional core may have different communications characteristics and requirements, but all may be described by appropriate annotation of the associated sockets. Such annotation is frequently simple to provide based on the aggregate characteristics of the algorithm implemented by the functional core, and can therefore be accurately modeled in advance of the existence of a new functional core. For instance, an MPEG2 video decoder could be modeled based primarily on the expected and worst-case read and write rates associated with compressed and decompressed MPEG macro-blocks and frames of decompressed video. By examining the composite actual or estimated characteristics of the various functional cores, the SoC architect can readily determine the overall throughput, latency, and quality of service (QoS) requirements for the active interconnect core. These performance requirements, coupled with the power and area goals for the finished SoC, drive the definition and implementation of the interconnect core.

Since the interconnect core is accomplishing most of the inter-core communications in the SoC, the interconnect core is likely to span the die. As has been mentioned previously, this means that the interconnect core is

likely to have internal delays dominated by wiring, rather than active logic. It is therefore essential that the SoC designer be able to predict the impact of wire delays, derived from an actual or estimated SoC floor plan, on the overall performance of the interconnect core. Such concerns are localized to the decoupled interconnect core, since the functional cores are each isolated into local regions where maximum wire delays are based on the physical extent of the functional core, and are therefore both smaller and more predictable than those inside the interconnect core. Long wire delay issues are resolved in the interconnect core, not the functional core.

7.1 Examples of Decoupled Interconnect Cores

Sonics, Inc. has been developing and marketing decoupled interconnect cores since 1997. Sonics has defined the term MicroNetwork as a heterogeneous integrated network that unifies, decouples and manages all communications between on-chip processors, memories, and I/O devices [15]. Sonics has introduced two distinctly different MicroNetworks into the marketplace. This chapter will use examples from these two MicroNetworks to illuminate some key optimizations enabled by decoupled interconnect architectures.

Sonics' SiliconBackplane MicroNetwork was first described in 1998 [16]. SiliconBackplane is designed to service moderate to high bandwidth requirements typical of SoCs that integrate three to ten processing-focused cores together with a similar number of streaming I/O interfaces and a few shared memory pools such as internal or external DRAM or SRAM. From a performance perspective, SiliconBackplane is optimized to service aggregate throughputs in the range of 300-4000 MBytes/sec.

Sonics3220 SMART Interconnect IP (S3220) was first described in 2002 [17]. S3220 is designed to service low to moderate bandwidth requirements typical of SoC subsystems that integrate several dozen peripherals together with links to a few control processors and DMA engines. From a performance perspective, S3220 is optimized to service aggregate throughputs in the range of 50-600 MBytes/sec.

Figure 6 shows a block diagram representing the use of two interconnect cores in an representative multimedia application. The arrows represent unique instances of the OCP socket. Arrows point from master to slave, indicating the direction of request flow. Data flow is normally bidirectional (i.e. read and write) across each socket.

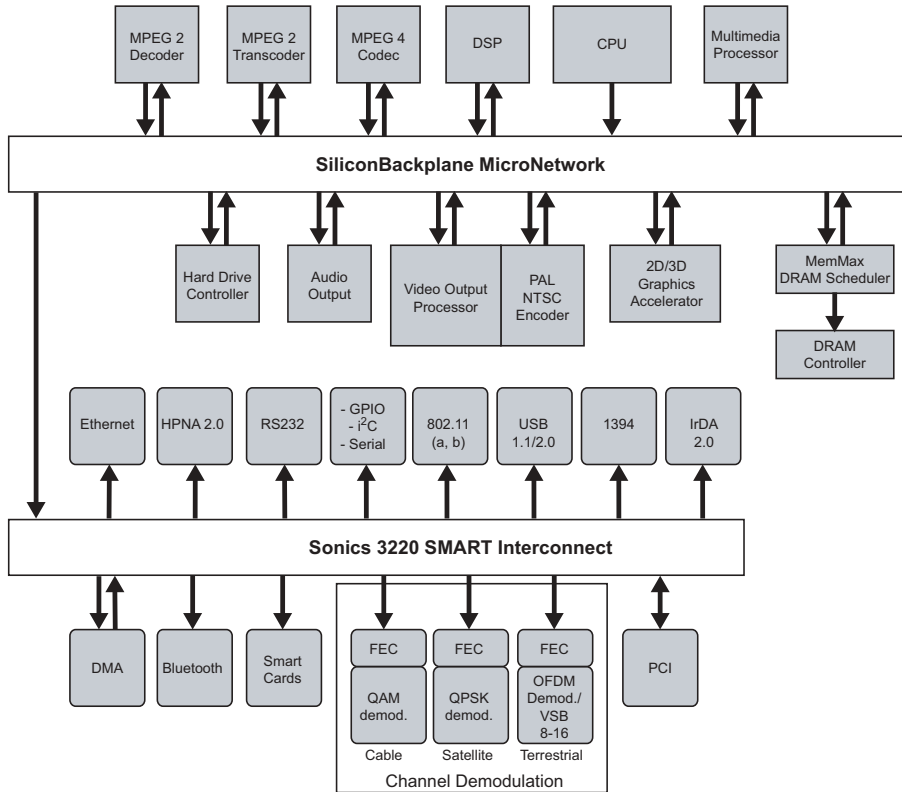


Figure 14-6. SiliconBackplane and S3220 interconnect cores in a digital set-top box

7.2 MicroNetwork Core Decoupling Capabilities

Both MicroNetworks extensively support the OCP socket for interfacing to functional cores. In addition, OCP is used as the MicroNetwork-MicroNetwork interface between separate instances of SiliconBackplane and/or S3220, as shown in Figure 6. Each MicroNetwork supports a rich subset of the total range of allowed OCP configurations [18].

Both MicroNetworks are composed of a set of *agents* that provide the active decoupling logic that bridges the functional core OCP interface(s) to the internal electrical, logical, and protocol environment of the MicroNetwork. Each agent is responsible for creating a local environment in which the attached functional core may operate as designed, isolating it from the different environments present at the other functional cores. This decoupling occurs at a number of layers. Figure 7 shows these layers, in

increasing levels of abstraction. The following subsections describe the agent decoupling capabilities in more detail.

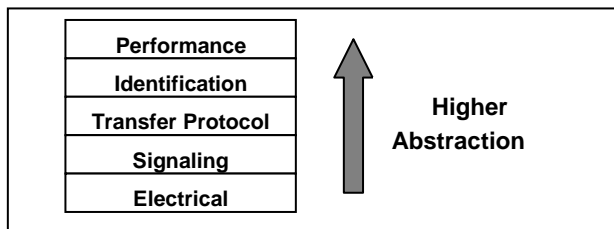


Figure 14-7. Decoupling abstraction layers for active interconnect cores

7.2.1 Electrical Layer Decoupling

The agent provides significant control over the degree of decoupling at the electrical layer. Functional cores may connect using an OCP clock frequency that is identical to the MicroNetwork, or at any integer divisor of the MicroNetwork clock. In addition, the OCP clock may be completely asynchronous to the MicroNetwork clock; in such cases, the agent includes the required synchronization logic to cross the clock boundaries. In all cases, the core sees simple, synchronous timing arcs.

The agents also have very flexible timing. For instance, if the core's outputs arrive late in the interface clock cycle, the agent must sample the signals into registers immediately. However, if the outputs arrive early in the clock cycle, the agent logic associated with the signal may be accomplished in the same cycle as crossing the interface. Thus, agents can cope with different core timings in an optimum fashion, which reduces die area and latency versus the conservative registered input and output approach [19].

The timing described at the electrical layer results from calculations derived from the transistor sizing on outputs, modeled as drive strength, the transistor sizing and internal cell routing associated with gate inputs, modeled as input capacitance, and the wiring lengths associated with connecting outputs to inputs, modeled as wiring capacitance and wiring delay. Accurate early predictions of the wiring induced delays associated with inter-core wiring are increasingly difficult to achieve as the level of integration grows. The MicroNetwork approach to this problem is simple: make these wires as short as possible. The MicroNetwork is specifically engineered to provide predictable mechanisms for spanning distance. As such, the agent should be placed adjacent to, or together with, the attached functional core, resulting in minimum wire length and, therefore, predictable interface timing and rapid timing convergence.

7.2.2 Signaling Layer Decoupling

At the signaling and transfer protocol layers, the agent relies on the OCP socket definition to supply the required interface configuration chosen by the functional core. The core developer should choose a configuration of OCP that closely matches the native characteristics of the core. The agent inherits the chosen configuration and is instantiated containing appropriate logic to handle the signaling and transfer protocol layers.

At the signaling level, the OCP supports several different data word sizes and flexible handshake-based flow control. Bridging different word sizes from the functional core into the MicroNetwork is automated by sequencing (i.e. packing and unpacking) logic built into the agent. Compliant state machine implementations of OCP may accept transfers unconditionally (i.e. independent from the current request), which simplifies timing analysis, or reactively (i.e. dependent on the current request), which offers the greatest design flexibility. Since the OCP uses only unidirectional signaling (i.e. all signals have precisely one driver), interface timing analysis is straightforward.

7.2.3 Transfer Protocol Layer Decoupling

OCP also supports a great degree of flexibility at the transfer protocol level. Compliant implementations offer choices over partial word transfers, burst protocols, request-response decoupling, request pipelining, and concurrency.

The agent implements appropriate request and data storage, plus appropriate sequencing logic, to interface the core-specific OCP configuration to the other agents in the MicroNetwork. Storage requirements are minimal for most agents; only those with proportionally high throughput requirements *and* significant mismatches between the core and MicroNetwork bandwidth require non-minimum storage.

7.2.4 Identification Layer Decoupling

In traditional data networks, source and destination identifiers are integrated in the packet headers, and are visible at many layers of the protocol stack. In contrast, OCP follows the traditional computing model of address-mapped reads and writes, so most identification is localized inside the MicroNetwork.

OCP includes an address field that the initiating core uses to identify the targeted core and intra-core resource. The MicroNetwork includes address-matching logic to implement targeted core selection. This logic is very

flexible, supporting positive and negative decoding, variable match width, and optional re-programmability to implement soft address maps. In addition, agents supporting initiating cores splice on source identifiers to core requests, allowing the identifiers to be fully specified at the system level, rather than forcing them to be compiled into the cores. The MicroNetwork uses source identifiers to accomplish thread mapping, selective connectivity, and response routing.

Only intra-core resource identification (i.e. partial address) is typically passed to the target core. Exceptions include bus bridges, where full addresses are typically required, and concurrent multi-bank memory controllers, where source identifiers are useful for establishing transaction priorities.

7.2.5 Performance Layer Decoupling

The agents work together in the MicroNetwork to implement performance-layer decoupling. Each agent may be configured with FIFO-like storage resources that are based on the performance required from the attached core for the application(s) hosted by the SoC. In addition, the MicroNetwork leverages its internal threading capabilities to time interleave multiple requests and thereby minimize buffering, while providing hardware guarantees of quality-of-service. This helps significantly in latency decoupling by minimizing latency uncertainty.

The cost of performance-layer decoupling varies greatly from agent to agent and application to application. The MicroNetwork-based design approach simplifies the calculation of the required storage, automates its creation, and allows simple re-tuning as refinement proceeds.

7.3 MicroNetwork Communication Optimization

The previous section has shown how the MicroNetwork is composed of a set of agents that provide a wide range of decoupling services to the attached functional cores. The same agents also implement the internal communication protocols of the MicroNetwork to enable agent-agent transactions.

The internal MicroNetwork protocols of SiliconBackplane and S3220 share several common characteristics. Given the dominance of wire delay over gate delay in advanced process technologies – particularly at the MicroNetwork level where the maximum distance between functional cores is often bounded by only the physical dimensions of the SoC die – both MicroNetworks leverage scalable internal pipelining so that the operating frequency of the MicroNetwork may be optimized. For instance, the

SiliconBackplane agents contain a number of optional internal pipelining points and bypass points.

Both SiliconBackplane and S3220 MicroNetworks use shared paths to transmit requests and responses. In this sense, they therefore implement a bus topology. A bus topology is useful because it minimizes routing area with respect to cross-bar and other switch topologies that offer parallel paths between initiators and targets.

Threads are supported internally in a similar fashion to the OCP socket. Threads allow the MicroNetwork to interleave requests and responses from multiple initiators at the granularity of single data words.

Transactions are non-blocking. Individual transfers may never occupy the shared inter-agent routing for more than a single pipelined cycle – regardless of target contention, bandwidth discontinuities, etc. Requests and responses attempt to transfer from a buffer in the sending agent to a buffer in the receiving agent.

The combination of threading and non-blocking mechanisms ensures that transactions between slow functional cores – regardless of their length or latency – never block transactions between higher bandwidth functional cores. In addition, access control (i.e. arbitration) is performed each cycle. This is feasible because threading allows arbitrary choice of which agent gets to send the next request and non-blocking guarantees that there will be a new access control opportunity on every cycle. This results in much higher transfer efficiencies than are normally associated with bus topologies [20] and also offers much tighter guarantees of QoS. For instance, commercial implementations of SiliconBackplane in telecom applications have achieved sustained throughputs above 85% of the ideal (data width * clock frequency) internal peak MicroNetwork bandwidth [21].

There are also several significant differences between the internal protocols of SiliconBackplane and S3220. This supports their different application focuses.

SiliconBackplane is optimized to support total internal bandwidths that are higher than the peak bandwidths of any attached OCP socket. In many applications, this allows SiliconBackplane to deliver cross-bar type bandwidths at the routing cost of a bus, much as a time-domain switch exploits *temporal* concurrency to deliver the same throughput as a space switch (which exploits *spatial* concurrency). In a traditional computer bus, it makes no sense to operate the bus at higher bandwidth than the highest-performance attached blocks. Even if the computer bus could be decoupled from the blocks, the routing savings associated with the bus would be lost in gate area spent on burst FIFO's to accomplish the decoupling. This is because computer buses arbitrate (and therefore block) on burst boundaries, so a slower block would need to store an entire burst into a FIFO at the

sending end, then arbitrate for access to the bus, and then move the data in a continuous burst into a FIFO at the receiver. The FIFO's are required because neither sender nor receiver can match the peak bus bandwidth.

SiliconBackplane uses very different internal protocols than a computer bus. The threaded/non-blocking internal fabric allows burst transactions to be interleaved with other traffic at a per-cycle basis. SiliconBackplane's access control scheme uses a combination of bandwidth pre-allocation via TDMA with unallocated and unused TDMA slots being dynamically allocated on a fair best-effort basis to waiting initiators as shown in Figure 8. The SiliconBackplane TDMA scheme is non-uniform so that the bandwidth guarantees may span a wide dynamic range, since the heterogeneous processing cores of a multimedia SoC may have orders of magnitude differences in throughput requirements. The designer implements this by the choice of the number of slots in the time wheel, and the number of slots allocated to each initiating agent. The TDMA scheme, when properly configured, allows transfers associated with *isochronous* or *quasi-isochronous* processing cores to be communicated across SiliconBackplane at the natural production and/or consumption rates of the cores. Such cores require very little buffering in the SiliconBackplane agents (merely enough to pack/unpack enough data to fill a SiliconBackplane data word), regardless of the transaction length/burst size. Thus, SiliconBackplanes protocols enable a bus topology interconnect to effectively use higher, decoupled internal bandwidth without requiring burst-deep FIFO's at ingress and egress.

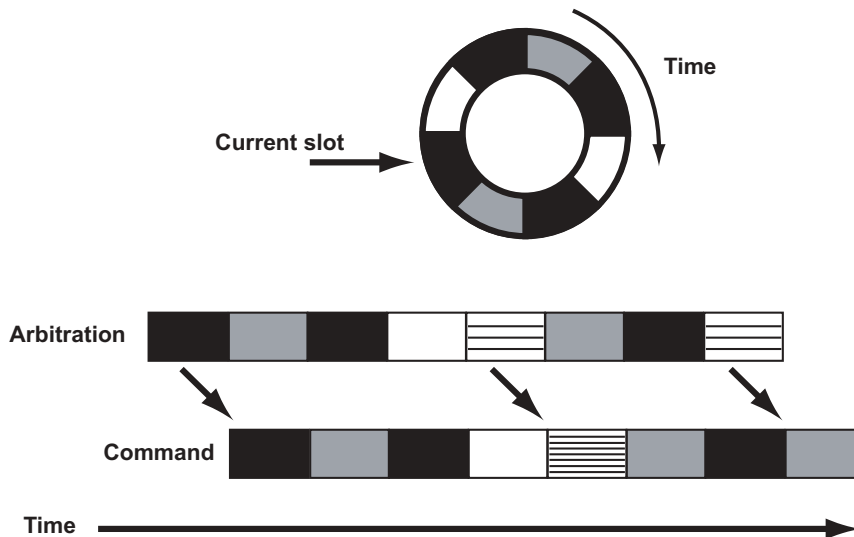


Figure 14-8. SiliconBackplane access control mechanism

Some SoCs (particularly in digital STB and HDTV applications) are characterized by inter-core data flows that mostly terminate in shared off-chip DRAM. In such cases, it is common that SiliconBackplane is configured with peak internal bandwidth equal to the peak DRAM bandwidth. With the overall SoC performance limited by the efficiency of the path to DRAM, it is essential that SiliconBackplane be able to saturate the DRAM channel. To accomplish this, SiliconBackplane is fully pipelined. This means that a single initiator agent to target agent transaction can fully occupy the interconnect, transferring a single SiliconBackplane (and thus DRAM) data word per cycle. However, the more normal case is that the DRAM subsystem is itself multi-threaded at the OCP socket. This allows multiple processing cores to interleave their request streams to the DRAM subsystem, enabling the DRAM controller to re-order DRAM commands so as to maximize the efficiency of the DRAM usage while guaranteeing QoS for real-time traffic. The benefits of such an approach have been described elsewhere [22].

The internal protocols of S3220 are optimized to efficiently connect peripherals spread around the pad ring of the SoC. Since the attached functional cores are typically numerous and small, optimization of die area per agent is crucial. S3220 implements pipelined internal protocols to ensure scalability in operating frequency while spanning large distances. However, full pipelining (as practiced in SiliconBackplane) can cost registers deep enough to cover the pipeline delay to be integrated into each agent, which would take too much area. Peripheral functional cores are rarely able to sustain full throughput, and are even less likely to require full throughput at the system level. Therefore, S3220 protocols are optimized to support a peak issue rate between each initiating thread and its target of one transfer every two cycles. This cuts the register area associated with covering the pipeline delay in half. Note that the multi-threaded, non-blocking nature of S3220 allows it to sustain a data transfer every cycle, as long as there are multiple initiating threads attempting to communicate with independent targets, as shown in Figure 9.

The figure shows a request 0 propagating from the initiating OCP to the targeted OCP, and the associated response 0 propagating from target to initiator. The internal pipeline cannot issue request 1 until the third cycle (as mentioned above), but the intervening grey cycles are available for independent initiator/target transfers.

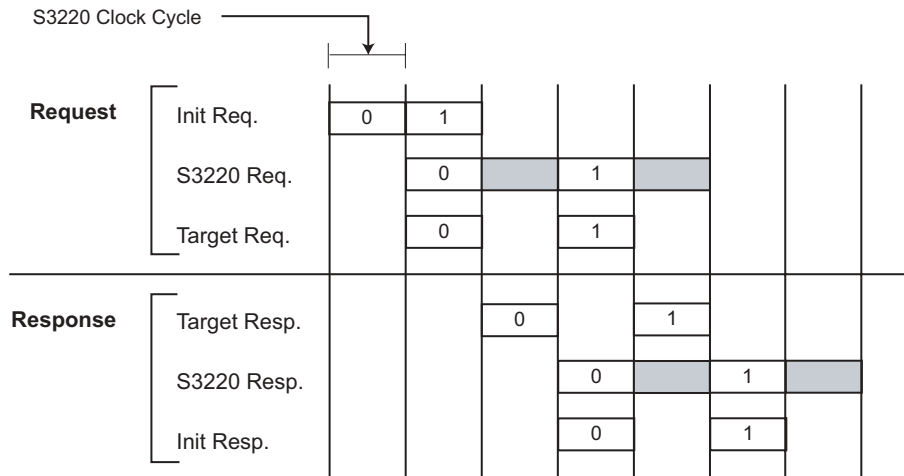


Figure 14-9. S3220 data transfer pipeline

Since the MicroNetwork must span the distance between all of the functional cores, they are constructed using logical structures that offer high performance, physically predictable timing results. Traditional on-chip bus topologies have migrated from distributed tri-state to centralized multiplexor implementations in order to accommodate the needs of ASIC-style design flows. However, the centralized multiplexor is essentially a “star” topology, with each sender driving an input across the SoC to the multiplexor, which generates both uncertain wiring delay (based upon the relative placement of the block and the multiplexor) and high routing congestion around the multiplexor – which is exactly what the designer intends to avoid by choosing a bus topology in the first place!

In contrast, both SiliconBackplane and S3220 implement a distributed multiplexor structured as a tree. As shown in Figure 10, each agent includes a transceiver that implements OR-tree multiplexing on the path towards the root of the tree, and a repeating buffer on the path back from the root. This simple structure has some very attractive physical properties. If the multiplexor tree is connected such that each agent connects directly to its nearest neighbors, then the length of the intermediate wires in the tree is closely related to the agent-agent spacing, which is in turn closely related to functional core-functional core spacing. Since most functional cores are fairly small, the inter-agent wires should be relatively short (1-2mm), and therefore the delay of these wires are unlikely to require additional repeater insertion. A properly-connected distributed multiplexor tree also minimizes routing congestion, since each logical bus wire is represented by only two physical wires at any point in the tree. Perhaps most important, the delay

through such a structure is quite predictable as soon as the number of agents and estimated SoC die size are known. This is because the timing behavior is dependent primarily upon the total wire length between the root and leaves of the tree and the depth of the tree. The tree depth is dependent on the number of functional cores. The total wire length is often estimated based on the MicroNetwork spanning the expected die. The evolving floor plan is then used to determine the multiplexor tree ordering, ensuring rapid timing convergence and low routing congestion.

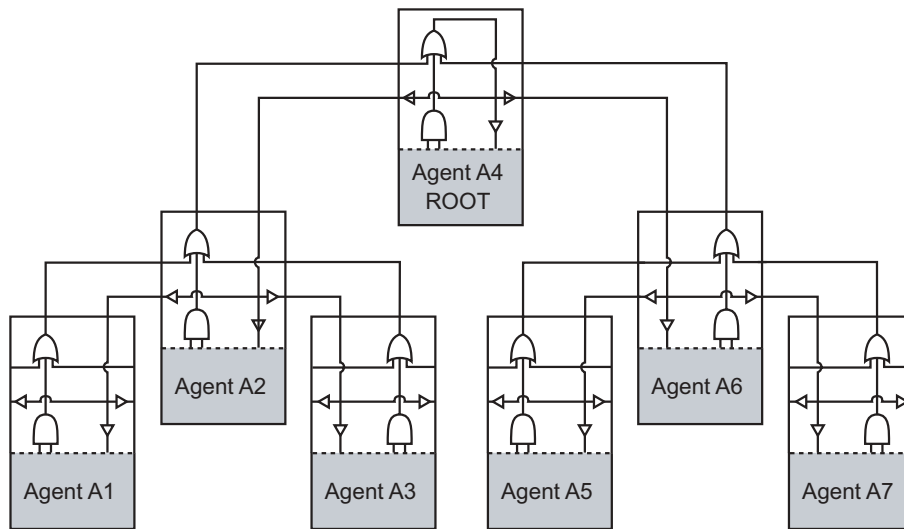


Figure 14-10. Internal MicroNetwork multiplexor tree

S3220 takes the physical layer optimization one step further. S3220 is optimized for low area and very low active and idle power. The designer may segment the S3220 logical bus into several physical branches, each of which contains a distributed multiplexor tree. A pipeline register at the root of the branches ensures that only the branch containing the agent servicing the targeted functional core is activated on a transfer request. This optimization avoids activating the other branches, which are likely to be long and thus have high capacitances that will significantly increase power dissipation.

7.4 Configuration and Generation of the MicroNetworks

To support the design of SoCs and MicroNetworks, Sonics has developed Sonics Studio, a graphical and command line SoC development environment. In developing a MicroNetwork instance, there are a large

number of options from which the SoC architect can choose and the Sonics Studio development environment provides a number of automation tools to make this task easier [23]. Each OCP socket can be configured independently and the MicroNetwork can be configured and tuned for a large number of performance and system management characteristics. While it is possible to create a system using just command line tools, it is more efficient to use a graphical interface to perform some of these tasks, with command line based tools and utilities available when needed.

The designer/architect can use the GUI to quickly create a working prototype of the SoC concept as shown in Figure 11. Each interconnect core is shown composed of a set of agents, which terminate one or more OCP interfaces (depicted as arrows). The square blocks represent actual functional cores, or models of cores, depending upon the design phase. The GUI automatically connects objects with compatible interface bundles; the bundle specifies the name and direction for each signal group for each type of interface that connects to the bundle. For example, the OCP interface bundle has different signal directions for Master versus Slave interfaces.

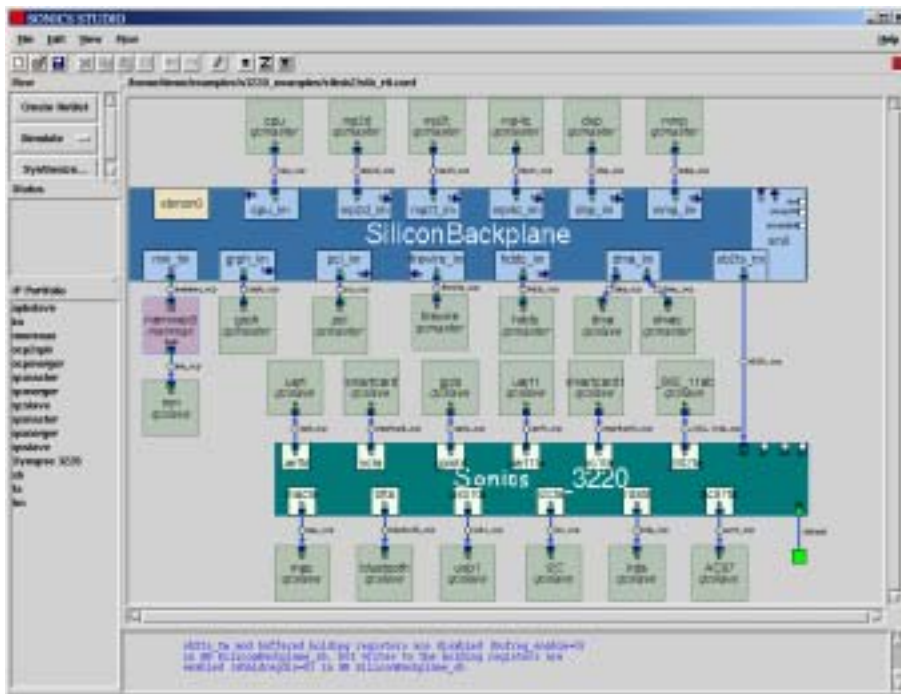


Figure 14-11. Sonics Studio GUI screen shot of behavioral digital STB design

To speed system design, Sonics Studio automatically configures MicroNetwork agents to inherit OCP configuration parameters when

attached to functional cores with OCP sockets. The GUI has configuration panes for each MicroNetwork for capturing parameters such as data path width and the characteristics of the arbitration system. For agent level parameters, there are settings for buffer depths, clock frequency ratios among others. These features streamline the MicroNetwork generation and configuration task.

Sonics Studio supports models in C++ or Verilog/VHDL languages. Behavioral models of both OCP masters and slaves are available for describing estimated functional core behavior. These models are configurable to meet the full OCP specification allowing them to be used to accurately model any OCP-compliant functional core at the Bus Functional Model level. Other tools include simulation monitors that capture trace data into ASCII files, disassemblers, protocol checkers, and performance measuring programs. These features facilitate the early data flow modeling that enable rapid performance analysis and optimization.

A finished MicroNetwork may have several hundred configuration settings. The source code for the MicroNetwork is compiled into the Sonics Studio RTL generator with configuration parameters captured from the GUI. The generator interprets the configuration parameters, computes context-sensitive default parameters, performs value checks, and passes the final parameters to a macro processor that configures the final RTL.

The generator also automatically produces HDL net lists to instantiate the MicroNetwork and the functional cores and to connect their interface bundles. This automation makes it possible to create SoC models quickly and to generate new models as the SoC is refined.

Several of the MicroNetwork configuration options affect pipeline optimizations to balance timing convergence versus latency. Timing models are available for agents in a wide range of configurations so that predictive timing is available as the design is optimized. An automated tool is used to generate configurations with estimated boundary timing constraints, to run the configured agent through logic synthesis and library mapping and to parse the static timing report to capture the results. This timing information is based on process technology, cell library and synthesis flow, so the timing and area information can be prepared before the architecture is finalized. This gives the architect accurate physical information to use in developing the SoC architecture.

Sonics Studio also provides interfaces to other tools such as simulation, design synthesis, floor planning, and timing analysis. By providing a central development environment that enables fast generation of MicroNetworks, rapid modifications of performance characteristics, integrated simulation, synthesis and verification tools, Sonics Studio empowers the SoC architect

with the tools needed to create complete SoCs using decoupled, active interconnect cores with standard interface sockets.

8. SUMMARY

Effective development of complex SoCs that incorporate many heterogeneous processing elements and tens to hundreds of I/O functions requires a new decoupled active interconnect methodology. Without socket-based decoupling and active interconnect cores, the process of design integration and verification breaks down at the inter-core communications level. Mismatching high-level functional core abstractions with low-level bus-oriented abstractions is the cause of this breakdown. The solution is decoupling the computational tasks of the functional cores from the communications functions. Inter-core and system-level communications functions should be placed in interconnect cores. For developing a decoupled interconnect-based system architecture, there are several potential standard solutions available including the OCP socket specification as a standard inter-core interface and the Sonics MicroNetwork family of interconnect cores. Using development tools such as Sonics Studio enables the design of complex SoCs based on sockets and interconnect cores, leading to the realization of “design by functional combination” approach to complex SoC design.

ACKNOWLEDGEMENTS

The theories expressed in this chapter, and the products and protocols described herein, have been defined and refined over the past seven years. Many people have contributed to this work. In particular, the author would like to acknowledge the SMART Interconnect IP development team at Sonics, Inc. for their creativity and dedication in pursuing these goals. In addition, the Working Group members of OCP-IP have helped enhance and focus these theories and practices. Finally, the author would like to thank Chris Bailey for his assistance in preparing this manuscript.

REFERENCES

- 1 C. Mead and L. Conway. *Introduction to VLSI Systems*. Addison-Wesley Publishing Company, Inc., 1980.
- 2 A. Jantsch and H. Tenhunen, *Networks on Chip*. Kluwer Academic Publishers, 2003.

- 3 D. Wingard, Tiles – An Architectural Abstraction for Platform-Based Design, EDA Vision, June 2002, available from www.edavision.com.
- 4 Mentor Graphics, February 2002.
- 5 G. Smith, Gartner Dataquest, Design Automation Conference, June 2003 as published in Electronics Engineering Times, June 9, 2003.
- 6 Open Core Protocol International Partnership, information from www.ocpip.org.
- 7 S. Kumar, On Packet Switched Networks for On-chip Communication. In A. Jantsch and H. Tenhunen, editors, *Networks on Chip*, Kluwer Academic Publishers, 2003.
- 8 ARM, Limited, AMBA Specification, Revision 2.0, May 1999, available from www.arm.com.
- 9 IBM, CoreConnect Bus Architecture, 1999, available from www.chips.ibm.com.
- 10 A. Doganis and J. Chen, Mentor Graphics Deep Submicron Technical Paper, Interconnect Simple, Accurate and Statistical Models Using On-chip Measurements for Calibration. July 1999.
- 11 Virtual Socket Interface Alliance. *Virtual Component Interface Standard*, OCB Specification 2, Version 1.0, March 2000., available from www.vsia.org.
- 12 Open Core Protocol Specification, available from www.ocpip.org.
- 13 W. Dally. Route Packets, Not Wires: On-Chip Interconnection Networks. In *Proc. of the 38th Design Automation Conference, June 2001*.
- 14 M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli. Addressing the system-on-a-chip interconnect woes through communications-based design. In *Proceedings of the 38th Design Automation Conference*, pages 203-211, 1998.
- 15 D. Wingard, MicroNetworks for Flexible SoC Platforms. In *Proc. of the Sophia Antipolis Forum on MicroElectronics (SAME2000)*, October 2000.
- 16 D. Wingard and A. Kurosawa. Integration Architecture for System-on-a-Chip Design. In *Proc. of the 1998 Custom Integrated Circuit Conference*, pp. 85-88, May 1998.
- 17 D. Maliniak, “Interconnect IP Steers SoC Integration into the Fast Lane,” *Electronic Design*, November 25, 2002.
- 18 SiliconBackplane MicroNetwork Overview, available from www.sonicsinc.com.
- 19 M. Keating and P. Bricaud. *Reuse Methodology Manual for System-on-Chip Designs*. Kluwer Academic Publishers, 1998.
- 20 H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, and L. Todd. *Surviving the SoC Revolution – A Guide to Platform-Based Design*. Kluwer Academic Publishers, 1999.
- 21 D. Wingard. Smarter Interconnects for Smarter Chips. Paper presented at Hot Chips Symposium on High Performance Chips, August 2002.
- 22 W. D. Weber, “Efficient Shared DRAM Subsystems for SoCs,” presented at Microprocessor Forum October 2003.
- 23 D. Wingard. MicroNetwork-based integration of SoCs. In *Proc. of the 38th Design Automation Conference, June 2001*.