MicroNetwork-Based Integration for SOCs

Drew Wingard
Sonics, Inc.
Mountain View, CA 94040
wingard@sonicsinc.com

Abstract

In this paper, we describe the concept of using an on-chip network as the fundamental communication architecture for a complex SOC design. We describe some of the salient features of a MicroNetwork that we have implemented, and introduce an automated development environment that makes MicroNetwork-based design productive. Finally, we show how one would use our MicroNetwork to integrate an SOC, and provide a general description of several completed MicroNetwork-based designs.

1. INTRODUCTION

Completing a design comprising several million logic gates is a mind-boggling task. The only feasible approach to such a design is to leverage the time-honored "divide and conquer" technique. In the world of System-on-Chip (SOC) design, a fundamental challenge is to structure the design process such that a team of engineers can execute the design in parallel. This challenge is magnified by both the lack of sufficient designers to complete a new design and the extreme market pressure to deliver a complex integrated circuit, together with software, in the time previously allocated to a printed circuit board-level system design.

The only available approach for most market segments is to rely heavily on re-use of previously defined semiconductor intellectual property cores (IP cores). The goal is to allow a small design team to leverage the work of others. Unfortunately, the reality is that the design team typically spends too much time re-working and re-verifying the IP cores, and not enough time focusing on the value-added functionality of the SOC itself. When coupled with the lack of predictability inherent in conventional communications architectures, the result is chip designs that fail to meet the performance, power, and area goals – much less the budgetary and time requirements.

In this paper, we introduce a new methodology for the design of complex SOCs that raises the level of abstraction available to the designer. This methodology leverages the capabilities of the MicroNetwork (an on-chip network) and a supporting development environment to enable true communications-based design. We start by presenting some of the related work in this field. We then describe the fundamental protocols and features of our MicroNetwork, and introduce the development environment that supports MicroNetwork-based integration. Finally, we describe how a designer uses our system to architect, implement, and verify a complex SOC.

38th DAC, June 18-22, 2001, Las Vegas, NV.

2. RELATED WORK

The vast majority of SOC designs that have been described to date are based upon IP cores stitched together using a mix of computer buses and various forms of point-to-point data or control links. Two commonly used on-chip computer buses are AMBA [1] and CoreConnect [2].

The notion of a data network on a chip is not new – many networking chips integrate a switch fabric to accomplish the routing function. Some have proposed similar communication architectures for SOCs [3]. Chang [4] provides a reasonable survey of the available communications topologies and protocols, and Lahiri [5] proposes a novel dynamic scheme for optimizing existing protocols.

Rowson [6] presents a compelling case for separating the functionality of an IP core from its communications. The Virtual Socket Interface Alliance's On-Chip Bus activity has produced a specification for an interconnect-independent data flow interface [7].

We attempt to address several shortcomings of these approaches. Computer bus-based communication architectures do not easily handle the real-time *isochronous* data flows associated with networking, telecommunications, and multimedia data streams. Crossbar topologies require too much area, particularly since many SOCs are dominated by the performance of a single memory subsystem (some form of DRAM). Routed architectures suffer from excessive latency and do not efficiently operate in SOC environments with wide variations in data flow characteristics.

All of the approaches focus only on the data communications challenge. However, the control flow and manufacturing test harness must also be integrated into a completed chip. Finally, no existing approach offers sufficient automation to facilitate accurate early system performance modeling, physical predictability, and rapid recovery from bugs or other late system changes.

3. MICRONETWORK OVERVIEW

We have devised and implemented an on-chip network to *manage* the communication between the IP cores on an SOC. We call this

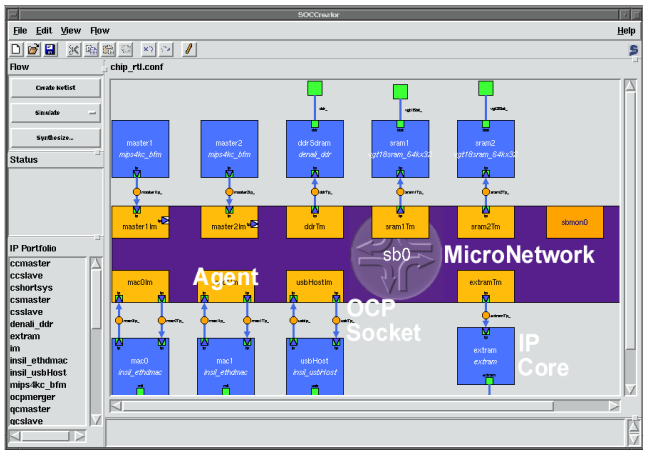

Figure 1. Block diagram of MicroNetwork from development environment GUI.

structure a MicroNetwork. A MicroNetwork is more similar to a data network than to computer buses. The MicroNetwork *unifies* the communication, which allows control flow and manufacturing test signaling to use the network alongside data flow. The MicroNetwork uses highly concurrent protocols, scalable data path widths, and pipelining to enable efficient unification. Furthermore, the MicroNetwork *decouples* the attached IP cores from the behavior of the network itself. This allows the IP cores to operate in their own local environment, accomplishing the separation of computation from communication. MicroNetwork decoupling greatly enhances the ability of design teams to work in parallel, since the MicroNetwork provides both structure and isolation. Data networks also manage, unify, and decouple.

A block diagram of our MicroNetwork system is shown in Figure 1. The system employs *agents* to actively manage each interface *socket* on behalf of the attached IP core. It is the collection of agents, together with the circuits and wires that connect the agents, that constitute the MicroNetwork. The next subsections describe the key features of the socket, the techniques that the MicroNetwork uses to manage the communication, and the physical structure of the MicroNetwork.

3.1 Socket

An important component of our system is a communication architecture-independent interface socket called OCP [8]. The OCP socket captures the interface behavior of an IP core at electrical, logical, protocol, and performance levels. OCP is architecture-independent for the same reasons as network sockets – to simplify the interface and to provide flexibility in how the abstracted network behaves.

Because the attached IP cores differ widely in communication needs, OCP is actually a family of interfaces. An IP core designer picks a member of the OCP family that matches the characteristics of the core. The family members differ in signal widths, the presence of certain signal groups, and the allowed encoding of some signals. As a result, an 8-bit UART will use a very different OCP member than an SDRAM controller with a 128-bit data word.

From a data flow perspective, OCP is very similar to the VSIA VCI [7]. However, OCP adds support for control signaling (such as interrupts) and manufacturing test signaling (such as internal scan).

3.2 Data Flow Support

While the fundamental data flow protocols used by our MicroNetwork have been previously described [9], we will summarize them here. We have chosen internal protocols for the MicroNetwork that are based upon hardware threads that can be time interleaved at a fine granularity. This allows the overall MicroNetwork throughput to be scaled up to meet the aggregate throughput required by the IP cores.

To enable highly scalable throughput, our MicroNetwork has parameterized internal data path widths and pipelining. The depth of the internal pipeline is chosen to minimize the achieved latency at the desired clock frequency. This pipeline depth is tuned in the development environment by selectively enabling pre-defined retiming registers inside the MicroNetwork.

Our MicroNetwork's access control (i.e. arbitration) scheme is based upon a rotating priority system based upon TDMA. On each clock cycle, the arbiter selects the agent that has been pre-allocated the time slot, unless the agent is idle. If idle, the time slot is made available via a round-robin scheme to agents that contend against one another for bandwidth.

Other key data flow features include designer-defined FIFO depths on a per-OCP basis, packing and unpacking support for mixed data path widths, and highly configurable address decoders.

3.3 Control Flow Support

Interrupts and other hardware-hardware control signals frequently cause logical and physical design problems due to their unstructured nature. Our MicroNetwork and its development environment provide the needed structure.

Control flow signals are usually captured as part of their OCP interface. These input and output signals are then either terminated in the agent (for signals accessed via a software-visible register) or routed across the MicroNetwork to the desired destination.

In all cases, the control flow signals are retimed in the agent. The retiming stage simplifies physical design and makes it simple to implement more complex error recovery and interrupt steering schemes.

3.4 Test/Debug Support

For those IP cores with test signaling specified in their OCP socket, the MicroNetwork can serve as the delivery vehicle for those test vectors. The designer is free to choose the number of scan chains and scan control signals.

In addition, our MicroNetwork has a debug port that can capture a complete trace of all the salient agent-agent communication. This scheme relies upon an external interface clock frequency that is several times as fast as the internal MicroNetwork frequency. The resulting sampled trace may be fed back to the analysis tools in the development environment to build a disassembled trace, run a protocol checker, or measure latency and bandwidth on a per-thread basis.

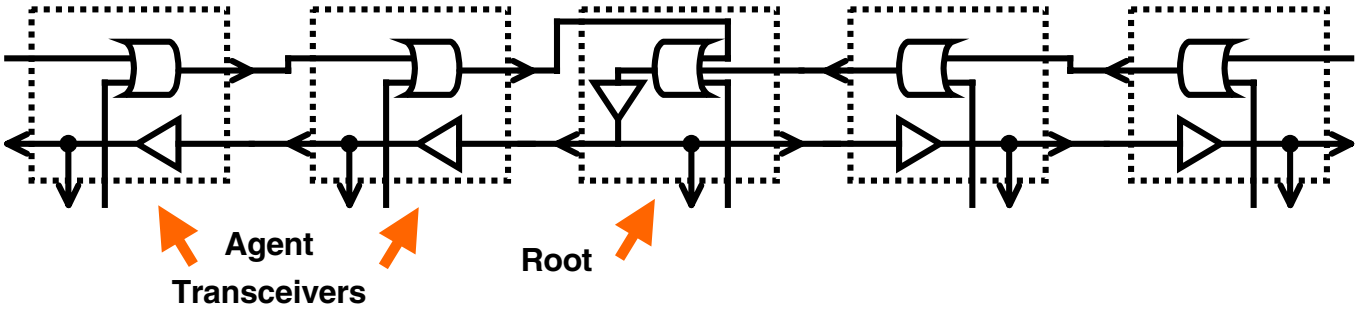

Figure 2. Physical implementation of MicroNetwork internal interconnect.

3.5 Physical Implementation

The logical topology of the internal MicroNetwork interconnect is a bus. Each signal driven onto the internal interconnect comes directly from a register, thus defining the beginning of a timing arc. The pipeline depth of the MicroNetwork determines the end of that arc. At low clock frequencies, a shallow pipeline allows the signals to propagate across the MicroNetwork interconnect, through the selected agent, and across the attached OCP socket. At high frequencies, a deep pipeline terminates the timing arc upon entry to the selected agent.

The MicroNetwork agents themselves are fairly small, self-contained units that are easy to implement physically due to the flexible pipelining. The MicroNetwork interconnect is formed by a group of transceivers that combine the multiplexing function of the bus with repeaters. As shown in Figure 2, a deep OR tree multiplexes the agent outputs while driving the signals to the middle. After the left and right sides are combined, the result is driven back to the agent inputs. The agents themselves are typically placed abutting their attached IP cores.

This structure has been shown to perform very well in physical design. First, placing the agent next to its IP core minimizes the wire length on the OCP nets, and therefore minimizes delay and wiring area. This is especially important for pre-characterized "hard" IP cores, where re-buffering the outputs is difficult.

Second, the total length of wiring required to implement the logical bus (without resorting to tri-state buffers) is only two times the length of the MicroNetwork. This assumes that the agents within the network are connected to their nearest neighbors from the SOC floor plan, and requires much less total wire than centralized multiplexing schemes.

Third, the delay of the structure is both small and predictable. In deep sub-micron designs, the delay of wires becomes dominant. At nominal IP core sizes of 20K to 200K gates, the expected agent-agent spacing is in the 1-3mm range. At those lengths, it is helpful to have a repeater in each agent transceiver – both in the forward and reverse direction. The overhead of a 2- or 3-input OR gate over a repeating buffer is very small (10ps or so), and this extra delay is easily compensated by the increase in predictability that arises from asking automated placement flows to solve simpler problems.

As an example, we fabricated a 9-agent MicroNetwork in a 0.18µm foundry process. With about 1 cm total span (length of MicroNetwork), the design completed at over 250MHz with a fully automated physical flow and a commercial standard cell library.

4. Development Environment

We have developed a commercial development environment around our MicroNetwork to enable designers to exploit the high degree of flexibility inherent in the MicroNetwork. We built the development environment using freely available tools. Most of the programs are in Python [10], with graphical interfaces linked to Tk [11]. All persistent data files are in ASCII, and most user-edited source files are Tcl [11]. The resulting development environment is simple to control from the command line or GUI, the embedded command language enables rapid assembly of complex systems, and all of the SOC-related source files are simple to manage.

The development environment itself is composed of a set of point tools, which are described in the next several subsections.

4.1 Block-based Design GUI

The example block diagram in Figure 1 is an actual screen shot from our block-based design GUI (SOCCreator). A user of this tool can rapidly assemble a working prototype of an SOC design concept. The GUI supports automated connection of objects with compatible interface *bundles*. An interface bundle is defined in a Tcl file that specifies the name and direction of each signal group, for each type of interface that connects to the bundle. For instance, the OCP interface bundle has different signal directions for Master-side versus Slave-side interfaces. Bundles allow the abstraction of complex interfaces into single lines in the GUI.

Designers spend more time configuring the MicroNetwork than connecting IP cores. To support this, the GUI has sets of configuration panels for the MicroNetwork. Some parameters (e.g. buffer depths, clock frequency ratios, etc.) are managed at the agent level because they are largely independent. Other parameters (e.g. the data path width and the arbitration system) are managed at the MicroNetwork level because of agent-agent dependencies.

The GUI also serves as a front end for most of the other point tools. Alternatively, the point tools may be initiated directly from the command line.

4.2 System Modeling

A first task in refining a communication architecture is to model the performance of the architecture. Our system includes several components to accelerate this process. MicroNetwork performance can be modeled in either C++ or Verilog/VHDL. The C++ model relies upon a C++ API view of OCP. It is possible to mix and match C++ and HDL models in a single simulation,

since there is an automated bridge between the OCP API and the simulator PLI.

The development environment includes behavioral master and slave models (both in C++ and HDL) that are frequently useful in modeling real or undersigned components as "black box" objects. The behavioral masters read an assembled trace that is produced from a transaction language. The transaction language is designed to be both a source form for simple tests, and an intermediate form to be output from cache modelers, trace generators, etc. The behavioral models are configurable to the full extent of OCP. This allows them to accurately model any OCP-compliant IP core at the "bus functional model" level.

The environment includes simulation monitors that capture traces into ASCII files, and disassemblers, protocol checkers, and performance measuring programs to characterize a simulation run.

4.3 RTL Generation

Our MicroNetwork is *extremely* configurable. A finished MicroNetwork has several hundred configuration settings. We compile the source code for the MicroNetwork into our RTL generator. The GUI captures configuration parameters into a Tcl file, which is fed to the RTL generator. The generator interprets the configuration parameter, computes context-sensitive default parameters, performs parameter value checks, and passes the final parameters to a macro processor that configures the final RTL.

The RTL generator also creates HDL net lists that instantiate the MicroNetwork and IP cores, and auto-connect their interface bundles. This step removes the tedious process of connecting dozens or hundreds of signal groups together in a text editor. This automation makes it practical to assemble SOC models throughout the design process, and to iterate those models as the SOC is refined.

4.4 Timing Characterization

Many of the configuration options in the MicroNetwork involve pipeline optimizations to balance timing convergence versus latency. It is therefore essential to have good models for the timing behavior of the MicroNetwork agents as the configuration changes. We have developed a tool that pre-characterizes the agents across a broad range of configurations. The tool generates configurations and estimated boundary timing constraints, runs the configured agent through logic synthesis and library mapping, and parses the static timing report to capture the results.

Because this timing information is based only upon the process technology, cell library, and synthesis flow, the timing information is normally prepared before the architecture is determined. This allows the designer to choose an architecture based upon accurate physical information.

4.5 Simulation Support

We have discussed the automated connectivity, behavioral models, and post-processing tools associated with the development environment. All of these tools are important during the simulation phases of the design. Additional automation supported by the development environment includes structure Tcl-based methods for capturing the steps required to prepare, execute, and analyze a simulation run. The IP core designer captures simple scripts for each phase, and the SOC integrator references those scripts to invoke the proper tool chain for each core at the proper moment in the simulation process.

4.6 Synthesis Support

The IP core designer normally captures pin-level timing characteristics together with their core. The attached agent inherits this information as timing constraints for synthesis. The development environment maintains a complete timing model for each boundary signal in the design (both inside the MicroNetwork and outside). The environment creates hierarchical synthesis scripts for each agent, propagating the timing values as constraints. The synthesis methodology encourages the specification of delays in a technology-independent fashion. Symbolic constants are then resolved once the technology-dependent parameters are determined.

5. USAGE MODEL

The previous sections provide a glimpse into a complex system. This section describes how designers actually use our MicroNetwork system to develop SOCs.

5.1 SOC Design Styles

In the conventional design style, each SOC design is fairly unique. Even small derivatives off a base architecture require full design and verification cycles.

There has been quite a bit of discussion in the electronics press about the benefits of design platforms for SOCs [4]. Most known examples are fairly *rigid*, in that the communications architecture is invariant across the uses of the platform. While this approach appears to work well in mature or low-performance market segments, it is unlikely that the rigid platform will satisfy the needs of markets where the list of IP cores for derivatives is unknown at the time of platform creation.

We believe that MicroNetwork design techniques offer a third approach – that of *flexible* SOC platforms [12]. In the flexible platform approach, we leverage the inherent scalability of the MicroNetwork and the automation of our design environment to gain the advantages of design platforms (e.g. stable software development environment, rapid physical design and verification of derivative designs) without the disadvantages (e.g. over-design for expansion, over-reliance on architect to guess derivative requirements).

5.2 SOC Integration Flow

Here is a typical SOC integration flow for MicroNetwork designers.

Step 1. Pre-characterize the MicroNetwork (Section 4.4).

Step 2. Determine base architecture.
Refine the SOC architecture to a block diagram showing the major components and principle data flows.

Step 3. Choose MicroNetwork data flow parameters.
Determine the desired peak bandwidth for the MicroNetwork based upon the principle data flows. One rule of thumb is to add the sum of the peak bandwidths of the real-time (isochronous) data flows together with the sum of the sustained bandwidth of the non-real-time data flows. Choose a MicroNetwork data path width and clock frequency to satisfy the desired peak bandwidth. Set the pipeline depths to balance latency versus physical design effort at the targeted clock frequency and estimated MicroNetwork length.

Step 4. Build data flow model.
Use supplied behavioral models or IP core models to construct a simulation model in the GUI. Build traces to represent principle data flows. Simulate and analyze the results. Iterate, if required.

Step 5. Improve the model.
Integrate more accurate IP core models, particularly around the memory subsystem. Begin allocating bandwidth to meet isochronous constraints. Consider performance benefits of exposing hardware threads at shared DRAM interface(s). Keep simulating.

Step 6. Test the physicals.
Synthesize, place, and route an early MicroNetwork-only net list to ensure physical predictability.

Step 7. Integrate IP cores and verify functionality.
Leverage behavioral models and core-specific simulation scripts for portable test bench.

Step 8. Map control flow.
Establish interrupt and error architectures. Route hardware-hardware signals, and terminate IP core control and status fields.

Step 9. Verify system functionality.
Map into hardware accelerator or emulator, as needed.

Step 10. Map manufacturing test and complete physical design.

Note that the automation of the development environment allows the design team to easily overlap architectural design with logical and physical design. Changes that previously required weeks of implementation and verification are now an afternoon's work. More importantly, the automation allows the designers to discover problems earlier in the design cycle, while the design is less firm and where improvements are simpler.

5.3 Example Data

Several commercial and test designs have been completed using our MicroNetwork. As of this writing, only one has been announced [13]. These designs are concentrated in premises and central office networking applications. All use a distributed DMA architecture, so even medium-speed peripherals have the ability to initiate an OCP request. The number of initiators per design ranges from 6 – 9. Most designs have at least two high-bandwidth memory subsystems.

The designs have been fabricated in 0.35, 0.25, and 0.18 µm CMOS technologies. They have MicroNetwork clock frequencies in the 80-250 MHz range. They have used internal data path widths of 32 and 64 bits, and pipeline depths of 2, 3, 4, and 5 MicroNetwork clock cycles.

6. CONCLUSION

We have presented a system that leverages the concept of an on-chip network to raise the level of abstraction at which an SOC can be designed. The flexibility of the MicroNetwork enabled us to create a commercial development environment that offers a high degree of design automation. The network and the environment together form a potent combination for successfully completing SOCs that consume several million logic gates.

We are barely scratching the surface of all the possible network-based communication architecture approaches that should be explored. We will certainly see more advanced protocols and topologies, as well as tighter integration with system-level design tools. There is also a strong opportunity for the application of formal methods to prove the performance and functionality of such complex systems.

7. ACKNOWLEDGMENTS

The author would like to acknowledge the invaluable contributions by the Sonics team to the theory and development of the SiliconBackplane MicroNetwork. In particular, the contributions of Scott Evans, Mike Meyer, Jay Tomlinson, and Wolf-Dietrich Weber merit special recognition.

8. REFERENCES

[1] ARM, Limited, *AMBA Specification*, Revision 2.0, May 1999, available from http://www.arm.com.

[2] IBM, *CoreConnect Bus Architecture*, 1999, available from http://www.chips.ibm.com/products/coreconnect.

[3] P. Guerrier *et al.*, "A Scalable Architecture for System-On-Chip Interconnections," In *Proc. of the Sophia Antipolis Forum on MicroElectronics (SAME '99)*, October 1999.

[4] H. Chang *et al.*, *Surviving the SOC Revolution: A Guide to Platform-Based Design*. Kluwer Academic Publishers, Norwood, MA, 1999.

[5] K. Lahiri *et al.*, "Communications Architecture Tuners: A Methodology for the Design of High-Performance Communication Architectures for Systems-on-Chip," in *Proc. of the 37th Design Automation Conference*, pp. 513-518, June 2000.

[6] J. Rowson and A. L. Sangiovanni-Vincentelli., "Interface-based Design," in *Proc. of the 34th Design Automation Conference*, pp. 178-183, June 1997.

[7] Virtual Socket Interface Alliance, *Virtual Component Interface Standard*, OCB Specification 2, Version 1.0, March 2000.

[8] Sonics, Inc., *Open Core Protocol Specification*, Version 1.0, October 1999, available from http://www.sonicsinc.com.

[9] 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.

[10] G. van Rossum, *Python Reference Manual*, Report CS-R9525. CWI, Amsterdam, April 1995.

[11] J. Ousterhout, *Tcl and the Tk Toolkit*. Addison-Wesley, Reading MA and London, April 1994.

[12] D. Wingard, "MicroNetworks for Flexible SOC Platforms," in *Proc. of the Sophia Antipolis Forum on MicroElectronics (SAME2000)*, October 2000.

[13] PMC-Sierra, Inc., *PM73140/PM73141 Voice Over Packet Processor Data Sheet*, Issue 1, August 2000.