A Network-on-Chip Monitoring Infrastructure for Communication-centric Debug of Embedded Multi-Processor SoCs

Bart Vermeulen[1], Kees Goossens[1,2]

[1]NXP Semiconductors Research, Eindhoven, The Netherlands, {Bart.Vermeulen,Kees.Goossens}@nxp.com

[2]Computer Engineering, Delft University of Technology, The Netherlands

Abstract— **Problems in a new System on Chip (SOC) consisting of hardware and embedded software often only show up when a silicon prototype of the chip is placed in its intended target environment and the application is executed. Traditionally, the debugging of embedded systems is difficult and time consuming because of the intrinsic lack of internal system observability and controlability in the target environment. Design for Debug (DfD) is the act of adding debug support to the design of a chip, in the realization that not every SOC is correct first time. DfD provides debug engineers with increased observability and controlability of the internal operation of an embedded system.**

In this paper, we present a monitoring infrastructure for multi-processor SOCs with a Network on Chip (NOC), and explain its application to performance analysis and debug. We describe how our monitors aid in the performance analysis and debug of the interactions of the embedded processors. We present a generic template for bus and router monitors, and show how they are instantiated at design time in our NOC design flow. We conclude this paper with details of their hardware cost.

I. Introduction

Modern process technologies enable the integration of a complex system on a single silicon die. Problems in a new design of such a System on Chip (SOC) often only show up when a silicon implementation of the chip is placed in its intended target environment and its embedded software is executed. These problems occur when the functional coverage of its pre-silicon verification was either unknowingly or necessarily incomplete [14].

The functional coverage of pre-silicon verification can be incomplete because a trade-off is made between the level of detail to include in a design model and the amount of compute time it subsequently takes to use this model in the verification of all relevant use cases. Detailed models are significantly slower to use for verification than more abstract models. As the market pressure restricts the time available to design and test a SOC, this necessarily also restricts the number of use cases that can be validated in exhaustive detail. As a result, human errors or bugs in tools or libraries may still slip through to prototype silicon and cause it to malfunction. To debug these errors using prototype silicon is difficult and time consuming because of the intrinsic lack of internal observability and controlability in the target environment. Design for Debug (DfD) [9] is the act of adding debug support to the design of a chip to increase its internal observability and controlability in a target environment. Improving these two abilities greatly facilitates finding the root cause of erroneous behaviour, both in time (i.e. when a deviation from the correct behaviour first occurs) and in space (i.e. which component(s), hardware and/or software, is (are) faulty).

Traditionally, debug methods and tools tend to focus on the computational part of a system, in particular, on the programmable Central Processing Unit (CPU) and its interaction with main memory. However many SOCs contain multiple control and digital signal processors, and a large part of the complexity resides in the interactions between these processors and other system components. In addition, design teams are starting to adopt a Network on Chip (NOC) as the on-chip communication backbone [3], possibly extended with local busses. Such a communication architecture presents a new range of debug challenges because its permits split, pipelined, and concurrent transactions between IP blocks on connections with differential Quality of Service (QoS).

Therefore a debug solution that covers the entire embedded system also has to include DfD to make the interactions between the IP blocks via the communication architecture observable and controlable. For this reason, we proposed to complement conventional computation-centric debug with communication-centric debug [8]. In this paper we focus on the debug monitoring of the communication. Debug control is discussed in [7], [18].

The remainder of this paper is organized as follows. We discuss the background of on-chip communication monitoring for functional debug and performance analysis in Section II. In Section III we identify possible monitoring locations in a SOC and its interconnect architecture and we define a generic monitor design template. Section IV describes the monitor instrumentation as part of the overall NOC design flow. Section V presents a break-down of the monitor hardware cost in its functional subcomponents. We conclude in Section VI.

II. Background

Debugging is a temporal and spatial refinement process. The root cause of erroneous behaviour needs to be located both in space and time. Traditionally only the design of the processors was extended with debug support, to observe and control the execution of the embedded software running on the processor. With the distribution of the computation across a SOC, and the resulting shift of the interaction (and potential interference) of processing threads to the communication architecture, debug support needs to be (deeply) embedded in the communication architecture as well. [2] and [12] present monitoring solutions based on bus-centric architectures. With the introduction of NOCs in SOCs, the scope of on-chip monitors has to be extended to include the NOC, as is e.g. done in [4] and [15]. Bus and network monitors become part of the on-chip debug hardware infrastructure, to observe the communications at the edge of or internal to the interconnect.

With these embedded monitors, the system can be viewed at the level of transactions. Inspecting transactions and detecting either missing transactions or transactions with incorrect attributes, enables a quick identification of a suspect master and suspect slave(s). Extending the debug scope further to include the communication infrastructure allows not only the identification of the suspect masters and slave(s), but also of a suspect path through the communication infrastructure. This identification allows for a large set of on-chip IP

blocks to be quickly discarded as the potential source of the problem, thereby greatly speeding up the debug process.

Debugging performance issues requires the embedded monitors to measure key system parameters in real-time. The performance metrics obtained from these monitors are subsequent correlated with the performance numbers estimated using simulation models. Using bus and network monitors to analyze a system's actual performance is important, because system communication resources are dimensioned at design time based on their estimated use in the system. When the (momentary) actual resource utilization is higher than expected, due to some unforeseen circumstances, this might lead to erroneous behaviour in the system. If the actual resource utilization is consistently lower than estimated, it indicates that the resources were overdimensioned, resulting in a higher Bill of Material (BOM) than strictly necessary. Being able to dimension the resources just right prevents over-design while providing the required QoS level.

III. MONITOR INFRASTRUCTURE

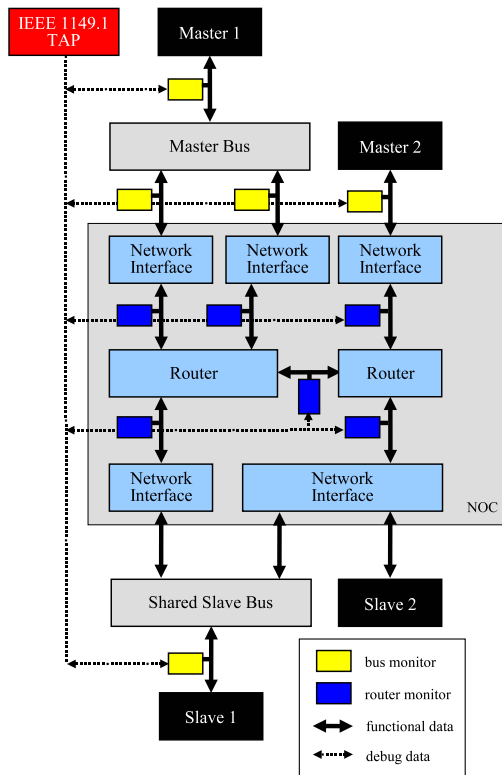Figure 1 shows the possible monitor locations in a SOC with a NOC.

Fig. 1. Network-on-Chip with Monitors.

Figure 1 shows that monitors can be inserted 1) on the master interfaces, 2) on the network interfaces, 3) inside the network on router interfaces, and 4) on the slave interfaces. They are programmed and queried either from a functional interface, or from a dedicated debug interface, such as the IEEE Std 1149.1-2001 Test Access Port (TAP) [11]. Figure 1 shows the latter option.

A monitor can provide a range of performance analysis and debugging functions. By using a generic monitor design template, each monitor in the system is optimized at design time to only include the necessary subset of components, thereby balancing the need for

debug and performance analysis support with the resulting additional hardware cost. Figure 2 shows a generalized monitor design template, which includes the following components.

1) *Protocol-Specific Front End*

Upon instantiation, the monitor is connected to a specific communication link. The sender and receiver on this communication link agree on a communication protocol to communicate data, e.g. the Advanced eXtensible Interface (AXI) [1] protocol or the Open Core Protocol (OCP) [13]. A Protocol-Specific Front End (PSFE) is used to decouple the other components of the monitor from the specifics of this communication protocol.

The PSFE includes optional transaction filters to restrict the collection of debug and performance metrics to a subset of the overall data communicated via the link. When required the PSFE can be instantiated with multiple transaction filters to enable filtering on different characteristics in parallel. Transaction filter may however also be implemented in the other monitor blocks. For a bus protocol, these filter criteria include: a) an address range, b) a reference data value, c) an associated mask value, and d) optionally a transaction ID identifying the source.

A router monitor observes the packetised data stream on a link between two routers or between a router and a network interface. Using knowledge of the NOC communication protocol, the PSFE can provide the raw data, and information on the End of Message (EOM) flag, the QoS of the data (Best Effort (BE) or Guaranteed Throughput (GT)), the word number in a NOC flit, and whether the data on the link belongs to a packet header, a packet body, or the end of a packet. The PSFE can be configured to use each of these characteristics to filter the transactions seen on the link.

2) *Bandwidth Utilization Measurements*

Bandwidth utilization is defined as the actual number of bus cycles used to transport data over a monitored link, normalized to the total number of bus cycles in a particular time interval. This metric is measured for all or a subset of traffic on the link, as determined by the programming of the optional transaction filter. An accumulator counts the number of bus cycles in which the link is actually used to transport data. A second accumulator counts the total number of bus cycles of the local communication link. Both metrics are reset and queried via the debug interface. The bandwidth utilization of the observed link is computed off-line by dividing the number of bus cycles actually used by the total number of bus cycles.

3) *Transaction Latency Measurements*

Latency measurements are useful for bus protocols, where a handshake is used to transfer each data element of a transaction. In a NOC data is typically transported across a link in a fixed amount of time, i.e. with no variation in latency. In those cases, it is not needed to include a transaction latency measurement component in the monitor.

A latency measurement block can store the most-recently measured transaction latency, the maximum latency, and the average latency. The average latency is obtained by first accumulating all latency samples and the number of transactions in a certain time interval separately. Dividing the former by the latter metric yields the average latency. An interrupt can be generated when a latency sample is greater than a certain, preprogrammed maximum.

4) Trigger Generation

A trigger block is used to generate a debug trigger after a pre-defined number of specific transactions have occurred on the monitored link. Filtering, as described above, can also be used here to specify the required traffic characteristics.

A counter is incremented for each match that occurs. Multiple trigger logic blocks are used to allow this to occur in parallel, e.g. for transactions in opposite directions. Communication links that allow overlapping bursts are supported up to the maximum number of pending transactions. This parameter has to be specified during the monitor's instantiation, as it requires additional buffering inside the monitor. The trigger output of the monitor is asserted when the counter reaches a preprogrammed trigger value. The monitor signals the trigger via its debug request and acknowledge ports to either the on-chip interrupt controller, or e.g. an on-chip cross-trigger architecture [2].

5) Checksum Calculation

Checksums are useful to calculate compact signatures for either the raw data on the communication link or abstracted protocol values. The same transaction filtering functionality is applied here. Whenever there is a relevant transaction, the Cyclic Redundancy Check (CRC) value is updated with the transaction's attributes. Cross-talk or other signal integrity issues may cause a CRC value to differ from one location on a communication path to another location or from a CRC value calculated off-line using an abstract system model (e.g. in SystemC). The comparison of CRC values calculated at multiple locations on a communication path enables the isolation of a suspect section of this path, thereby speeding up the debug process.

6) Control and Status Registers

The monitor control and status registers are accessible from the debug interface. Through these registers an engineer can program and query the performance metrics to measure, and define trigger points to stop the system at.
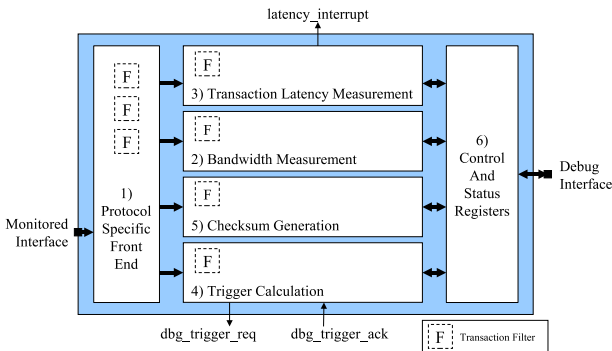

Fig. 2. Monitor Design Template.

IV. DESIGN FLOW

The NOC hardware is generated and instrumented with debug support hardware using an automated design flow [5], [6] (Refer to Figure 3).
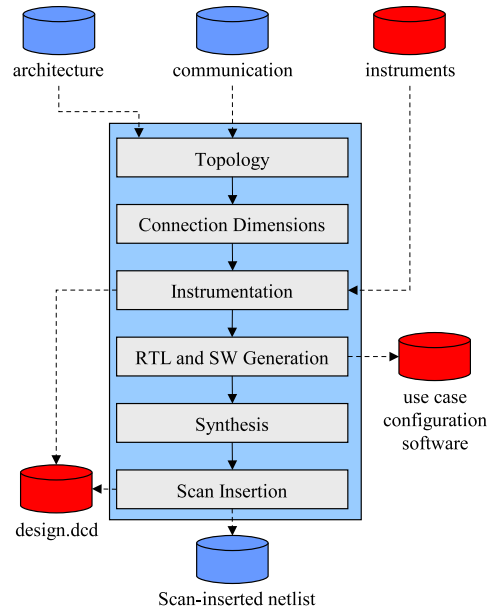

Fig. 3. Monitors in the Æthereal Design Flow.

Given an architecture specification containing a description of the IP blocks to be connected to the NOC, their interface ports with their associated communication protocol, and a set of communication use cases (i.e. sets of concurrent applications), an application-specific network topology is created and its resources are dimensioned such that they can handle the communication requirements for all use cases.

The resulting network is then instrumented based on the user's instrumentation specification. For the monitors the user can specify at which locations in the communication architecture the monitors are added (refer to Figure 1. For each monitor, the user can also specify which debug and performance analysis functions are included. Depending on the communication protocol that is associated with the link that each monitor is connected to, the flow will instantiate an appropriate PSFE. The outputs of this step in the flow are 1) an application-specific NOC with fully-specified connections between masters, local buses, network interfaces, routers, monitors, and slaves, and 2) a database for the debugger software with details of the debug support added to the SOC.

Afterwards the Register Transfer Level (RTL) files of the NOC instance are generated. In addition, embedded configuration software is generated to configure the NOC at run time for each use case. Commercial tools are subsequently used to synthesize the NOC RTL and insert scan chains [8], [17]. The scan insertion tools output scan chain listing information, which is used to complement the information in the proprietary Debug Chain Database (DCD) file, which permits debugger software to associate the bits in the debug scan chains with the bits in the RTL registers of the IPs and NOC components [7]. This data abstraction step offers a key benefit to design engineers by allowing the behaviour of the system to be analyzed using design views they are familiar with [10], [16].

V. EXPERIMENTAL RESULTS

To evaluate the hardware area cost of the monitors and their main contributors, we synthesised a small set of our bus and router monitors using a commercial synthesis tool and an industrial-quality 65nm CMOS standard cell library, and obtained detailed area logs.

The monitor area cost numbers presented below are expressed in NAND2 equivalent gates, to decouple them from the specific process technology used.

Table I shows the break-down of the area cost of a 64-bit AXI bus monitor that can handle up to 8 pending transactions, and transaction IDs up to 4 bits. Each function has its own transaction filter implementation (Filter A, Filter B, Filter C).

Function	G.E.	Fraction
Trigger, CRC & Filter A	17,618	48.29%
PSFE, Control and Status registers	8,440	23.13%
Latency & Filter B	7,744	21.22%
Bandwidth & Filter C	2,684	7.36%
Total	**36,486**	**100.00%**

TABLE I

64-BIT AXI MONITOR AREA DISTRIBUTION PER FUNCTION.

Table I indicates that the largest part of the monitor is used for the transaction trigger calculation. For the AXI protocol, its Filter logic has to keep track of up to 8 pending transactions, and support slave-side address calculation to match each data element. As such, it has an internal First-In First-Out (FIFO) that scales with the number of pending transactions, and the width of the transaction ID. The PSFE, Control and Status registers also consume a large part of the total area, to store all configuration data for the monitor.

Table II shows the break-down of the area cost of a 32-bit AXI bus monitor that can handle up to 8 pending transactions, and transaction IDs up to 4 bits. Each function again has its own transaction filter implementation (Filter A, Filter B, Filter C).

Function	G.E.	Fraction
Trigger, CRC & Filter A	9,773	37.34%
PSFE, Control and Status registers	7,130	27.23%
Latency & Filter B	7,062	26.98%
Bandwidth & Filter C	2,211	8.45%
Total	**26,175**	**100.00%**

TABLE II

32-BIT AXI MONITOR AREA DISTRIBUTION PER FUNCTION.

The 32-bit AXI monitor obviously requires less logic to implement, in particular because the buffering requirements are less, and the bus data elements it has to operate on are smaller. Table III shows the break-down of the area cost of a router monitor that can handle a link width up to 32 bits, and flit sizes up to 3 words. All functions utilize the same transaction filter implementation (Filter), located in the PSFE.

Function	G.E.	Fraction
PSFE, Control and Status Registers & Filter	9,508	72.12%
Bandwidth	1,437	10.90%
CRC	1,202	9.12%
Trigger	1,035	7.86%
Total	**13,182**	**100.00%**

TABLE III

ROUTER MONITOR AREA DISTRIBUTION PER FUNCTION.

The router monitor requires far less area to implement due to three main factors: 1) The router monitor does not implement the latency measurement functionality, as was explained in Section III, 2) The functional blocks reuse the same filter, which permits amortizing its area cost over all blocks, reducing the total area cost of implementation, at the cost of imposing a restriction on the end user, and 3) the protocol on a router link is simpler, and does not require any pending transactions or slave-side addresses to be tracked, eliminating the cost of a storage FIFO and an address calculation unit.

VI. CONCLUSION AND FUTURE WORK

In this paper, we presented a monitoring infrastructure for multi-processor SOCs with a NOC, and explained its application to performance analysis and debug. We presented a generic template for bus and router monitors, and show how they are instantiated at design time by our NOC design flow. Experimental results show that the required area cost for each monitor is relatively small compared to a million-gate SOC design, enabling their liberal use at strategic places throughout a SOC communication architecture to help find functional errors and assist in real-time performance analysis.

REFERENCES

[1] ARM. *AMBA AXI Protocol Specification*, 2003.
[2] ARM Limited. CoreSight System Design Guide. http://www.arm.com.
[3] L. Benini and G. De Micheli. Networks on chips: a new soc paradigm. *Computer*, 35(1):70–78, 2002.
[4] Călin Ciordaş, Twan Basten, Andrei Rădulescu, Kees Goossens, and Jef van Meerbergen. An event-based monitoring service for networks on chip. *ACM Transactions on Design Automation of Electronic Systems*, 10(4):702–723, 2005.
[5] Călin Ciordaş, Andreas Hansson, Kees Goossens, and Twan Basten. A monitoring-aware network-on-chip design flow. *J. Syst. Archit.*, 54(3-4):397–410, 2008.
[6] Kees Goossens, John Dielissen, Om Prakash Gangwal, Santiago González Pestana, Andrei Rădulescu, and Edwin Rijpkema. A Design Flow for Application-Specific Networks on Chip with Guaranteed Performance to Accelerate SOC Design and Verification. In *Proc. DATE*, pages 1182–1187. IEEE Computer Society, 2005.
[7] Kees Goossens, Bart Vermeulen, and Ashkan Beyranvand Nejad. A High-Level Debug Environment for Communication-Centric Debug. In *Proc. DATE*, 2009.
[8] Kees Goossens, Bart Vermeulen, Remco van Steeden, and Martijn Bennebroek. Transaction-Based Communication-Centric Debug. In *Proc. Int'l Symposium on Networks on Chip (NOCS)*, pages 95–106. IEEE Computer Society, 2007.
[9] A.B.T. Hopkins and K.D. McDonald-Maier. Debug support for Complex Systems on-Chip: A review. *IEE Proceedings Computers and Digital Techniques*, 153(4):197–207, 2006.
[10] Y. Hsu, B. Tabbara, Y. Chen, and F. Tsai. Advanced techniques for rtl debugging. In *Proc. DAC*, pages 362–367, 2003.
[11] IEEE JTAG 1149.1-2001 Std. *IEEE Standard Test Access Port and Boundary-Scan Architecture*. IEEE Computer Society, 2001.
[12] Rick Leatherman and Neal Stollon. An Embedded Debugging Architecture for SoCs. *IEEE Potentials*, 24(1):12–16, 2005.
[13] OCP International Partnership. *Open Core Protocol Specification. 2.0 Release Candidate*, 2003.
[14] Bill Roberts. The verities of verification. *Electronics Design, Strategy, News (EDN)*, 2003.
[15] S. Tang and Qiang Xu. A multi-core debug platform for noc-based systems. In *Proc. DATE*, pages 1–6, 2007.
[16] B. Vermeulen, Y-C. Hsu, and R. Ruiz. Silicon debug. *Test and Measurement World*, pages 41–45, 2006.
[17] Bart Vermeulen, Kees Goossens, and Siddharth Umrani. Debugging Distributed-Shared-Memory Communication at Multiple Granularities in Networks on Chip. In *Proc. Int'l Symposium on Networks on Chip (NOCS)*, pages 3–12, 2008.
[18] Bart Vermeulen, Kees Goossens, Remco van Steeden, and Martijn Bennebroek. Communication-centric SOC debug using transactions. In *Proc. European Test Symposium (ETS)*, pages 69–76. IEEE Computer Society, 2007.