A Multi-Core Debug Platform for NoC-Based Systems

Shan Tang[†‡] and Qiang Xu[†]

[†] Department of Computer Science & Engineering
The Chinese University of Hong Kong, Shatin, N.T., Hong Kong

[‡] Broadband Communication Network Lab
Beijing University of Posts & Telecommunications, Beijing, China
Email: {tangs,qxu}@cse.cuhk.edu.hk

Abstract

Network-on-Chip (NoC) is generally regarded as the most promising solution for the future on-chip communication scheme in gigascale integrated circuits. As traditional debug architecture for bus-based systems is not readily applicable to identify bugs in NoC-based systems, in this paper, we present a novel debug platform that supports concurrent debug access to the cores under debug (CUDs) and the NoC in a unified architecture. By introducing core-level debug probes in between the CUDs and their network interfaces and a system-level debug agent controlled by an off-chip multi-core debug controller, the proposed debug platform provides in-depth analysis features for NoC-based systems, such as NoC transaction analysis, multi-core cross-triggering and global synchronized timestamping. Therefore, the proposed solution is expected to facilitate the designers to identify bugs in NoC-based systems more effectively and efficiently. Experimental results show that the design-for-debug cost for the proposed technique in terms of area and traffic requirements is moderate[1].

1 Introduction

With the ever advancement in semiconductor technology, designers now are able to integrate an entire system onto a single chip, known as system-on-a-chip (SoC). To meet stringent time-to-market requirements, the development of SoC devices is based on the design reuse philosophy, i.e., they are created by combining tens or even hundreds of pre-designed IP cores and custom user-defined logic (UDL) together. Most existing SoCs utilize on-chip buses to connect these IP cores in a "plug-and-play" fashion that mimic off-chip board-based systems. However, it is well known that on-chip buses do not scale well with the shrinking technology feature size, in terms of both operational speed and power dissipation. Therefore, the concept of network-on-chip (NoC) [5, 9] was proposed recently and it is generally regarded as the most promising solution for the future on-chip communication scheme in gigascale SoCs. A number of NoC structures have been proposed since then, as surveyed in [6]. Although different in one way or another, they all contain three fundamental components: network interfaces (NIs) that connect the IP cores to the NoC, routers that transport data between NIs according to pre-defined protocol, and links that connect routers and provide the raw bandwidth. For a typical NoC (e.g., [10]), the NIs convert transaction messages into packets by chopping them into pieces and adding header that contains routing information to every piece; packets are further split into flits (the smallest transfer unit) in order to reduce packet latency with wormhole routing strategy [1].

At the same time, because of the high design complexity and the inaccurate abstracted models used in various design phases, existing verification techniques, such as simulation, formal verification, static timing analysis, and emulation cannot guarantee the correctness of the first silicon [11, 18]. Since time-to-market dictates the success of a chip, a silicon debug strategy that helps identifying bugs effectively and efficiently is of crucial importance. Silicon debug support, however, requires to increase the controllability and observability of the design's internal node to a level that is much higher than what manufacturing test generally requires [11, 22]. While capturing snapshots through JTAG run-control interface provide basic postmortem debuggability and are widely utilized in practice [20], the trend is to embed more design-for-debug (DfD) structures for hardware tracing difficult-to-find bugs (e.g., [2, 4, 12, 13, 21]). With the above techniques, debugging a single embedded core is a relatively well studied problem (still challenging though). However, since embedded cores communicate with each other during normal operation, debugging one core at a time can be ineffective and sometimes misleading [17], especially for SoCs with a number of processors. To tackle this problem, a multi-core debug solution was proposed for bus-based SoCs by introducing various on-chip instrumentation (OCI) blocks customized for diverse processors, logic cores and embedded buses [12, 17].

The above bus-based debug architecture, however, is not readily applicable for NoC-based systems, as debugging NoC-based systems introduces extra difficulties with the much more complex communication schemes. The authors in [8] introduced an event-monitoring service for the NoC that

[1]This work was supported in part by the Hong Kong SAR UGC Direct Grant 2050366 and Hong Kong SAR RGC Earmarked Research Grant 2150503.

provides continuous observability for the NoC transactions. Their technique, however, requires the NoC to be substantially over-designed in order not to affect the monitored user connections' bandwidth. In addition, effective debugging should get all the active parties involved, i.e., the activities on both the NoC and the cores under debug (CUDs) need to be traced in a unified architecture, which is not addressed in their work.

In this paper, we propose a novel debug platform for NoC-based systems. The main contributions of our proposed architecture are:

- We introduce debug probes in between the CUDs and the NIs, which not only control and monitor the CUDs through the cores' debug interface (e.g., JTAG) without introducing dedicated wires to connect to the system debug interface, but also support effective inter-core transaction analysis. By building connections between multiple debug probes and a system-level debug agent connected to the chip-level JTAG interface, the proposed architecture provides concurrent debug access to multiple cores and their transactions.
- We develop a novel "two-pass" debug strategy, which, together with our configurable cross-triggering and timestamping mechanism, controlled by an off-chip debug controller, facilitates the designers to synchronize multiple DPs' debug operations and qualify trace data, thus dramatically increasing the design's debuggability.

The remainder of this paper is organized as follows. Section 2 reviews the related work in this domain. The proposed debug platform for NoC-based systems are detailed in Section 3. Next, Section 4 analyzes the cost of the proposed debug architecture in terms of area and NoC traffic. Finally, Section 5 concludes this paper.

2 Prior Work and Motivation

Today's complex SoCs usually need to go through one or more re-spins to become bug-free even though half of the system development effort is allocated to verification tasks [16]. This is because all the verification techniques are applied to a model of the chip instead of the actual silicon and usually cannot be applied exhaustively [20]. Therefore, efficient and effective post-silicon validation techniques need to be developed to reduce time-to-market.

Debugging silicon (as a blend of hardware and software), is an extremely complex problem and cannot be tackled without effectively observing the operations of the design's internal nodes. Designers traditionally rely on postmortem debugging approach that captures a snapshot of the SoC through run control interface to find bugs [20]. This technique is quite effective in identifying those easy-to-find bugs that leave "evidences" when the SoC halts, but fails to find deeper bugs because these bugs manifest themselves only after a long time. In addition, postmortem debugging is also not applicable for mission-critical applications, such as automotive engine management, as it is an intrusive technique. Therefore, a more effective debugging strategy for a complex SoC is to monitor and trace it during normal operation [11]. The monitored data can be either stored in an on-chip tracing memory or transferred out of the chip via a trace port. Processor cores have started to use traces to facilitate software debug for some time [4, 13], in which instruction flows and sometimes data accesses are logged so that designers are able to reconstruct the program flow to find errors. With the increasing complexity of logic cores, it is important to trace some of their internal signals as well [2, 21]. Moreover, the interactions between CUDs should also be logged to achieve a better visibility of the system. Such huge amount of trace data, however, are difficult to analyze. Trace qualification through event-based triggers is therefore extremely important to reduce trace volume down to only the required information.

A debug platform for complex SoCs should provide the following features: concurrent debug access to the CUD's debug interfaces (e.g., JTAG) and their communication structures with limited chip input/output (I/O) requirement, system-level triggering and trace, and debug event synchronization for cores from multiple clock domains [17]. Concurrent debug access to multiple embedded cores has been addressed in [17, 19] by forming JTAG chains and introducing dedicated DFT logic to let it conform the IEEE 1149.1 Standard. In addition, the authors in [17] implemented a so-called HyperDebug module in their debug architecture that connects all CUDs to handle multiple cross-triggering. A reference clock is also utilized in the HyperDebug module for global timestamp synchronization in their debug architecture.

The bus-based debug architecture in [17], however, is not readily applicable for NoC-based systems. First of all, effective NoC transaction monitoring and analysis is a challenging problem itself. In addition, the debug architecture in [17] incurs large routing overhead with all CUDs connecting to the HyperDebug module. This problem is exacerbated when multiple cores need to transfer their traced data off-chip in real-time, which may be necessary for complex SoCs. In NoC-based systems, cores are able to communicate with each other through the NoC already. Therefore, it is not necessary to route dedicated wires to implement cross-triggering and transfer out traced data.

There is also limited work in debugging NoC-based systems, focusing on monitoring the NoC communication structure instead of the entire system. [7] introduced an event-monitoring service for the NoC, in which dedicated monitoring probes are attached to routers and provide basic observability of the NoC in the form of bits. Designers, however, usually prefer to work at a higher abstraction level during debug phase, i.e., at the transaction level instead of at the bit level when debugging inter-core communication. As a result, the same authors redesigned their monitoring probe later to be able to monitor transactions [8]. However, because their monitoring probes are attached to the routers and have no knowledge about the start of a transaction message, special message identification method is developed, which, unfortunately, not only depends on the message structure, but also requires the monitors enabled before the actual monitored connection is set up. The large amount of continuous observed data thus require the

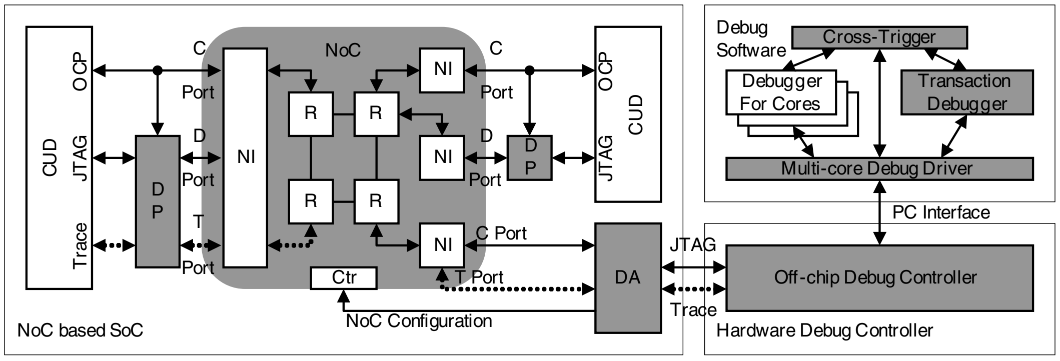

Figure 1. Proposed Debug Platform for NoC-based Systems

NoC to be substantially over-designed, in order not to affect the monitored user connections' bandwidth. In addition to the NoC hardware cost, this blindly monitoring strategy is also not very effective from a debug point of view, as bugs are usually easier to be found with qualified traces triggered under thoughtful conditions.

The above observations motivate us to develop a novel debug platform suitable for NoC-based systems, as shown in the following section.

3 Proposed Debug Platform

With the ever-increasing difficulty in the post-silicon validation process, many today's embedded cores come with some DfD structures (e.g., [2, 4, 12, 13, 21]), usually controlled through JTAG interface. Some complex cores also comprise a trace port to transfer debug data outside of the chip for further analysis in real-time. Under such circumstances, we propose a debug platform for NoC-based systems as shown in Fig. 1. It is composed of three main parts: the *on-chip debug architecture* that provides concurrent debug access to the embedded cores and the NoC, the *supporting debug software* (e.g., various debuggers for different cores), and the *off-chip debug controller* that provides a physical translation layer between the on-chip debug architecture and the software debug tools. For the ease of discussion, we assume the cores in the NoC-based systems communicate with each other using the OCP protocol [14]. Please note, however, our debug platform works with the other transaction-based communication protocol (e.g., AXI [3] or DTL [15]) as well.

In this paper, we mainly focus on the design of the on-chip debug architecture and the off-chip debug controller, which together form the hardware infrastructure of the proposed debug platform. Before presenting them in detail, let us briefly discuss how the supporting debug software fits in the platform. The supporting debug software provides the graphical user interface (GUI) and/or command line window to control the debug process and display the debug information. It contains three layers. In the middle are the various core specific debuggers from core providers (e.g., ARM [4]) and a transaction debugger that controls and observes the transactions between embedded cores. A cross debugger at the upper layer, which communicates to multiple CUD's debuggers and the transaction debugger at the same time, is in charge of debugging the interrelated operations of multiple cores. At the bottom layer is a multi-core debug driver, which forwards the debug requests and collects debug information to/from the off-chip debug controller, through a PC peripheral interface (e.g., parallel interface or ethernet). Since multiple debug requests/information can arrive the driver asynchronously, the debug driver needs to sort them and add/remove addressing labels to/from them.

3.1 Overview

Unlike [17], in which all CUDs are connected together with dedicated wires for multi-core cross debugging, in our proposed debug platform, the debug commands and debug data are transferred by reusing the NoC with guaranteed service on throughput and latency, together with the functional data. Consequently, the proposed debug platform for NoC-based systems significantly reduces the routing overhead when compared to [17]. This debug solution is also scalable with the increasing amount of debug tracing data that come from multiple CUDs at the same time, since we can simply increase the NoC bandwidth used for debug purpose.

Reusing the NoC for debug command and debug data transfer, however, also results in some difficulties because of the latency from a CUD's debug interface to the system debug interface. This is relatively easy to handle for single core debug since we can build a virtual connection between the CUD and its software debugger that looks transparent to the debugger. For multi-core cross debugging, however, it is often important to synchronize the triggering events and operations of multiple cores. For example, the designer may want to stop the system and observe the status of several cores when one core hits a triggering condition. Since the debug commands are transferred to different CUDs with different NoC transfer latencies (although predictable, as discussed in Section 3.4), we propose a *two-pass debug strategy* with the help of a global synchronized timestamping strategy controlled by the off-chip debug controller. The basic idea is to log the trigger event happening time in the first pass, and then to restart the system and control all the CUDs to be triggered simultaneously with this timing information in the second pass.

To implement the above two-pass debug strategy, a so-called *timestamping register* is introduced for every CUD to add timestamp to its operations (e.g., trigger events). In addition, every CUD is equipped with a *delay counter* to store the number of clock cycles that it should start to execute the debug

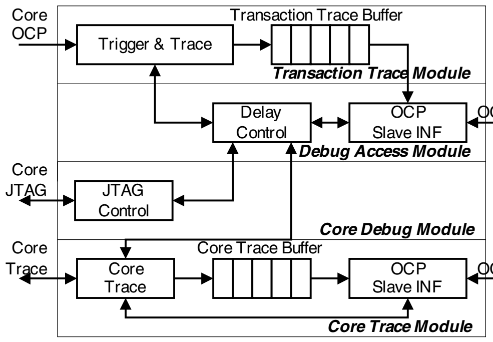

Figure 2. Debug Probe Block Diagram

command after receiving it. The distributed timestamping registers in all the CUDs are initialized to be zero synchronously when the debug process starts by the off-chip debug controller. This is achieved by writing the "reset timestamping register" debug commands to the CUDs with different pre-calculated values to the delay counters of the CUDs. These debug commands arrive different CUDs at different time, but since they start to execute the commands only after the values in their delay counters is reduced down to zero, these CUDs are able to operate synchronously with appropriate pre-calculated delay counter values. After the synchronous reset, these timestamping registers start to accumulate themselves at their own clock rates. Whenever an event is trigged in a cross debugging environment, the value in the timestamping register is logged and sent out to the debug controller. Then in the second pass, the debug controller can determine how to control all the involved cores so that they can operate synchronously.

3.2 On-Chip Debug Architecture

As a system-level debug solution for NoC-based systems, the on-chip debug architecture needs to be able to concurrently access interacting cores through their JTAG interfaces. In addition, the NoC transactions also need to be effectively observed through the debug architecture. At the same time, because today's SoC devices are often pin-limited, it is preferable to reuse the system-level JTAG interface for debug purpose (plus multi-bit trace port, if necessary). To achieve the above goals, our proposed on-chip debug architecture mainly contains the following components:

- core-level debug probes (DPs) in between every CUD and its network interface;
- a system-level debug agent (DA) controlled by the off-chip debug controller through JTAG interface;

With quality of service (QoS) guaranteed NoC channels between the DPs and the DA, we can map all debug resources (e.g., timestamping registers, delay counters, and various debug control and status registers) to memory-mapped registers as the slave of the DA and access them in a unified manner.

Debug probe: The block diagram of a typical DP is shown in Fig. 2. From the debug point of view, embedded CUDs have three kinds of interfaces: the functional communication port (OCP interface here), debug interface (usually JTAG) and the optional trace port (e.g., for processors [4, 13]). Therefore, at the CUD side, the role of the DP is also three-fold: (i) it generates the necessary JTAG signals to control and observe the CUD; (ii) it traces the OCP transactions with reconfigurable trigger conditions; and (iii) it transfers the trace data that come from the CUD off-chip for further analysis. As shown in Fig. 2, the debug probe is composed of four main components:

- *transaction trace module*, which monitors transactions between CUDs and records them based on pre-defined trigger conditions and recording rules;
- *debug access module*, which uses a standard OCP slave interface as the service access point of the DP;
- *core debug module*, which controls and observes the CUD through its debug interface (usually JTAG) by translating the debug register read/write commands into the CUD's debug access protocol;
- *core trace module*, which controls the CUDs' trace port (if any), buffers the traced data and provides a OCP slave interface for the DA to read out the buffered data through NoC connection.

At the NoC side, the debug resources are accessed through a OCP slave interface (denoted as the *D port*). In addition, an optional DP trace port can be implemented (denoted as the *T port*). These interfaces can share the same NI with the CUD's communication port or use dedicated NI.

Debug agent: The debug agent contains a test access port (TAP) controller that receives debug commands and sends debug data from/to the off-chip debug controller through the chip-level JTAG interface. In order to control and observe the debug resources in the CUDs and DPs, as discussed above, all these debug resources are memory-mapped into a unified address space and we introduce an extra JTAG instruction "DEBUG_REG" and the corresponding JTAG data register "DEBUG_REG_CMD" to access them with the following format:

Field	**Description**
WR/RD	'1' for write operation; '0' for read operation
ADDR	Register address
DATA	Register write/read data
READ_VALID	'1' means read data available

The off-chip debug controller writes and reads debug registers with a procedure described as follows.

Off-Chip Debug Controller	**Debug Agent**
Write 'DEBUG_REG' command into IR;	Use DEBUG_REG_CMD register as DR;
Write operation	
Shift in 'write' command to DR: :WR:ADDR:DATA:-:	Write specified register;
Read operation	
Shift in 'read' command to DR: :RD:ADDR:-:-: Do {Shift out the contents of DR;} While (READ_VALID) != '1'; The DATA field in RD is the valid data;	Read specified register; Set READ_VALID when data is ready;

Note: To realize a delay write operation, the pre-calculated delay value must be written to the DP's delay counter first with the above method.

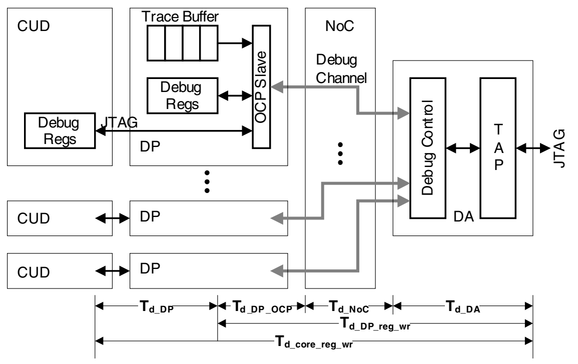

Figure 3. Debug Access Delay

The DA implements an OCP master interface to transport the debug information to and from the DPs through NoC connections. In addition, the DA also supports the optional chip-level trace port controlled by the off-chip debug controller, so that the software debuggers are able to access the trace data. If the NoC connections can be reconfigured on the fly (e.g., in [10]), we can equip the DA with NoC reconfigurability so that the debug connections between the DA and some DPs can be constructed dynamically to make better use of the NoC bandwidth. We are also able to deactivate the debug infrastructure in normal operational mode to improve the performance and save the power consumption of the NoC-based system. If the NoC connections need to be fixed during the design phase, however, we have to build debug connections between all DPs and the DA in the first place.

3.3 Off-Chip Debug Controller

Our off-chip debug controller serves as a translation layer between the software debuggers and the on-chip debug architecture, which builds transparent connections from the debuggers to the CUDs and DPs. It receives the debug commands from multiple debuggers, schedules them and then controls the on-chip debug agent through the system-level JTAG interface. One of the main duties of the debug controller is to cross debug multiple DPs and embedded cores simultaneously. To achieve this objective, the delay to read/write the distributed debug resources must be predictable. This can be seen from Fig. 3, which demonstrates an abstracted model of the debug access delay. The delay to write a debug register in a DP i is:

$$T_{d\_DP\_reg\_wr}(i) = T_{d\_DA}(i) + T_{d\_NoC}(i) + T_{d\_DP\_OCP}(i) \qquad (1)$$

, while the delay to write a debug register in an embedded core i is:

$$T_{d\_core\_reg\_wr}(i) = T_{d\_DA}(i) + T_{d\_NoC}(i) + T_{d\_DP\_OCP}(i) + T_{d\_DP}(i) \qquad (2)$$

, in which $T_{d\_DA}(i)$, $T_{d\_NoC}(i)$, $T_{d\_DP\_OCP}(i)$, and $T_{d\_DP}(i)$ represent the DA processing delay, the NoC transfer latency, the OCP slave processing delay, and the processing delay between the DP and the CUD's JTAG interface when the DA write debug registers to the DP or core i, respectively. Clearly, $T_{d\_DP\_OCP}(i)$ and $T_{d\_DP}(i)$ are fixed with the pre-defined communication protocol. $T_{d\_NoC}(i)$ is also a fixed value for debug channels with guaranteed service. $T_{d\_DA}(i)$ can be also determined although it depends on the debug register width[2].

[2]For the sake of simplicity, we use fixed-width debug registers (32 bits) in our current implementation.

While writing multiple debug registers at the same time in the DPs and/or the CUDs from the DA corresponds to a "one-to-many" relationship and hence is easier to control, reading multiple debug registers simultaneously is troublesome because the "many-to-one" connections result in unpredictable processing delay in the DA. Therefore, reading debug registers from the DP's OCP slave interface is done sequentially in our design to make its delay predictable.

3.4 A Cross-Triggering Example

Let us use an example to show our multi-core cross debug process in this section. Consider the following debug scenario that involves one DP and three embedded cores: the DP detects a NoC transaction that meets its trigger condition, and the designers would like to stop the three cores at the same time so that their internal states can be observed and analyzed. The debug process is as follows:

```
1. set DP trigger condition;
2. start DP and cores at the same time;
3. while (!triggered) check trigger status;
4. read out timestamp of the trigger event;
5. calculate relative trigger time for each core;
6. reset debug environment;
7. start DP and cores at the same time;
8. stop cores at trigger time;
9. hand over the control to the debuggers;
```

Among the above tasks, the most difficult one is to start/stop different cores at the same time, which is done by writing their debug registers with different delays. That is, since the register writing delays, $T_{d\_core\_reg\_wrA}$, $T_{d\_core\_reg\_wrB}$ and $T_{d\_core\_reg\_wrC}$, can be pre-calculated, we intentionally insert different delays, T_{delayA}, T_{delayB} and T_{delayC} to the corresponding DPs so that all write register commands take effect at the same time, as shown in Figure 4.

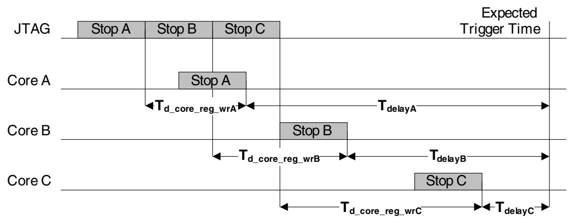

Figure 4. Synchronous Debug Control

4 Cost Analysis

The proposed scalable debug platform for NoC-based systems also brings some DfD cost, as analyzed in this section.

4.1 Area Cost

As introduced in Section 3, the on-chip debug architecture contains a number of debug probes and one debug agent. Therefore, the area of the DP should be well controlled to reduce the total DfD area cost. We implement an experimental DP design in a commercial 90nm CMOS library. A detector for comparing OCP commands and addresses and a transaction analyzer that detects the read delay are instantiated in our design. The transaction trace buffer used in the design is an

32×128-bits asynchronous FIFO, in which we can store 32 transaction records, including 2 bits of the record identifier, 48 bits of the timestamp, 78 bits of the payload. We also implement a 32×32-bits core trace buffer. These two buffers are implemented by general-purpose flip-flops in our current implementation. The total area of this design is 188059 μm^2 (about 57k gates). Further analysis shows that the two trace buffers occupy most of the area in the DP (as expected), more than 80 percent. If using SRAMs as FIFO, the area of these buffers can be reduced significantly. The control logic inside the DP is about 10k gates, which is considered to be acceptable for giga-scale NoC-based systems. Adding more triggers and core trace modules will not increase the area of the control logic significantly. For the debug agent, since it is an interfacing module in our current implementation without trace buffer, its size is quite small, less than 5k gates.

4.2 NoC Traffic Cost

As the proposed solution utilizes NoC connections to transfer debug commands and debug data, the NoC needs to be over-designed to accommodate the debug traffic. Generally speaking, more debug connections and/or higher debug bandwidth requirements per DP result in more DfD area for the NoC. Since only a small number of DPs/CUDs need to be debugged concurrently at a time in most cases, the proposed solution for reconfigurable NoCs is more area-efficient when compared to the NoCs without such capabilities.

For the debug traffic, all DPs share a chip level JTAG debug port. Given the clock rate of JTAG, $f_{TCK}\ Hz$, its throughput is $B_{JTAG} = \frac{1}{f_{TCK}}\ bps$. Due to the processing overhead in the off-chip debug controller and the debug agent, the actual usable bandwidth is much smaller, $B_{actual} = \frac{N_{data}}{N_{overhead}+N_{data}} B_{JTAG}\ bps$, in which N_{data} and $N_{overehead}$ represent the cycles used to transfer actual debug data and the other data, respectively. Therefore, the bandwidth of the NoC debug connection between every DP and the DA should be B_{actual}. For the debug register read/write operation, we need to shift in the JTAG instruction first and then the debug register address and the actual debug data at last. The actual utilized debug bandwidth B_{actual} is approximately $\frac{B_{JTAG}}{3}\ bps$ if we are able to read/write the debug registers of DPs/CUDs continuously.

For a particular DP, the NoC debug connection is only used at the moment that the DA is accessing it. Therefore, given the number of debug connections N_{dc}, the maximum NoC debug channel utilization rate is $R_{max} = \frac{1}{N_{dc}}$. While the NoC debug channel utilization rate for the write operation can almost reach R_{max} as we are able to write multiple DPs continuously; it is lower for the read operation as we need to wait between reads from different DPs.

5 Conclusion

In this paper, we present a novel debug platform for NoC-based systems, which, by using the NoC channels to transfer debug commands and debug data, supports concurrent debug access to the cores under debug (CUDs) and the NoC in a unified architecture. The proposed solution introduces core-level debug probes between the CUDs and their network interfaces and a system-level debug agent controlled by an off-chip multi-core debug controller. It provides in-depth analysis features for NoC-based systems, such as NoC transaction analysis, multi-core cross-triggering and global synchronized timestamping. As a result, it is expected to facilitate the designers to identify bugs in NoC-based systems more effectively and efficiently, at the cost of moderate DfD area.

References

[1] K. M. Al-Tawil, M. Abd-El-Barr, and F. Ashraf. A Survey and Comparison of Wormhole Routing Techniques in a Mesh Networks. *IEEE Network*, 11(2):38–45, Mar-Apr 1997.

[2] Altera Inc. Design Debugging Using the SignalTap II Embedded Logic Analyzer. http://www.altera.com.

[3] ARM Ltd. AMBA AXI Protocol Specification. http://www.arm.com.

[4] ARM Ltd. How CoreSight Technology Gets Higher Performance, More Reliable Product to Market Quicker. http://www.arm.com.

[5] L. Benini and G. de Micheli. Networks on chips: A new SoC paradigm. *Computer*, 12(1):70–78, 2002.

[6] T. Bjerregaard and S. Mahadevan. A survey of research and practices of network-on-chip. *ACM Comput. Surv.*, 38(1):1–54, 2006.

[7] C. Ciordaş et al. An event-based monitoring service for networks on chip. *ACM TODAES*, 10(4):702–723, 2005.

[8] C. Ciordaş et al. Transaction Monitoring in Networks on Chip: The On-Chip Run-Time Perspective. In *Proc. IEEE IES*, 2006.

[9] W. J. Dally and B. Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. In *Proc. DAC*, pp. 18–22, 2001.

[10] K. Goossens, J. Dielissen, and A. Rădulescu. The Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design & Test of Computers*, 22(5):21–31, 2005.

[11] A. B. T. Hopkins and K. D. McDonald-Maier. Debug Support for Complex Systems on-Chip: A Review. *IEE PCDT*, 153(4):197–207, 2006.

[12] R. Leatherman and N. Stollon. An Embedded Debugging Architecture for SOCs. *IEEE Potentials*, 24(1):12–16, 2005.

[13] MIPS Technologies Inc. EJTAG Trace Control Block Specification. http://www.mips.com.

[14] OCP International Partnership. Open Core Protocol Specification. http://www.ocpip.org.

[15] Philips Semiconductors. Device Transaction Level (DTL) Protocol Specification. http://www.philips.com.

[16] Semiconductor Industry Association (SIA). *The International Technology Roadmap for Semiconductors (ITRS): 2003 Edition.*

[17] N. Stollon et al. Multi-Core Embedded Debug for Structured ASIC Systems. In *Proc. DesignCon*, 2004.

[18] B. Vermeulen and S. K. Goel. Design for Debug: Catching Design Errors in Digital Chips. *IEEE Design & Test of Computers*, 19(3):35–43, 2002.

[19] B. Vermeulen, T. Waayers, and S. Bakker. IEEE 1149.1-Compliant Access Architecture for Multiple Core Debug on Digital System Chips. In *Proc. ITC*, pp. 55–63, 2002.

[20] B. Vermeulen, T. Waayers, and S. K. Goel. Core-Based Scan Architecture for Silicon Debug. In *Proc. ITC*, pp. 638–647, 2002.

[21] Xilinx Inc. Chipscope Pro Software and Cores User Guide. http://www.xilinx.com.

[22] Q. Xu and N. Nicolici. Resource-Constrained System-on-a-Chip Test: A Survey. *IEE PCDT*, 152(1):67–81, 2005.