A Debug Probe for Concurrently Debugging Multiple Embedded Cores and Inter-Core Transactions in NoC-Based Systems

Shan Tang and Qiang Xu
Department of Computer Science & Engineering
The Chinese University of Hong Kong, Shatin, N.T., Hong Kong
Email: {tangs,qxu}@cse.cuhk.edu.hk

ABSTRACT

Existing SoC debug techniques mainly target bus-based systems. They are not readily applicable to the emerging system that use Network-on-Chip (NoC) as on-chip communication scheme. In this paper, we present the detailed design of a novel debug probe (DP) inserted between the core under debug (CUD) and the NoC. With embedded configurable triggers, delay control and timestamping mechanism, the proposed DP is very effective for inter-core transaction analysis as well as controlling embedded cores' debug processes. Experimental results show the functionalities of the proposed DP and its area overhead[1].

1. INTRODUCTION

Existing system-on-a-chips (SoCs) usually utilize on-chip buses to connect embedded cores. However, it is well known that functional bus does not scale well with the shrinking technology feature size, because of its speed limitation and high power consumption when a large number of cores are attached to the bus. Network-on-Chip (NoC), as a promising solution for integrating IPs in the future giga-scale SoCs, is gaining wide acceptance in both academia and industry (e.g., [5, 6, 9]). A typical NoC contains three parts: network interfaces (NIs) that connect the IP cores to the NoC and convert transaction messages into packets, routers that transport packets between NIs according to a predefined protocol, and links that connect routers and provide the raw bandwidth.

While the NoC facilitates designers to integrate more IP cores on-chip, the verification problems become more challenging and it is likely that the NoC-based systems need to go through one or several re-spins to become bug-free. An efficient and effective silicon debug strategy for such complex systems is therefore of crucial importance. Existing silicon debug techniques (e.g., [4, 12, 18]), however, mainly target bus-based systems and are not readily applicable to NoC-based systems with a totally different and more complex communication scheme.

In this paper, we show the detailed design of a novel debug probe (DP), one of the key elements of the debug platform for NoC-based systems presented in [19]. The proposed DP, inserted between every core under debug (CUD) and the NI associated to it, not only supports effective inter-core transaction tracing and analysis, but also provides full debug access to the CUD through its debug interface (e.g., JTAG interface). With embedded programmable triggers, delay control and timestamping mechanism, the DP facilitates multi-core concurrent debug with cross triggering and real-time tracing capabilities. Moreover, the proposed DP uses the NoC connections to transport debug and trace data instead of using dedicated wires, which fits better in the NoC environment. When compared to the monitoring probes attached to the NoC routers presented in [7, 8] that monitor the NoC only, the proposed DP is able to debug and trace the activities on both the NoC and the CUD concurrently at the system-level.

[1]This work was supported in part by the Hong Kong SAR RGC Earmarked Research Grants 2150503 and 2150558.

The remainder of this paper is organized as follows. Section 2 and Section 3 overview related work and the NoC debug platform presented in [19]. The proposed DP design is detailed in Section 4. Section 5 gives the simulation results and analyzes the area cost of the debug probe. Finally, Section 6 concludes this paper.

2. PRIOR WORK

Debugging an SoC (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 [11]. While capturing snapshots through JTAG interface provides basic postmortem debuggability and are widely utilized in practice [21], the trend is to embed more design-for-debug (DfD) logics for hardware tracing difficult-to-find bugs (e.g., [1, 3, 12, 13, 22]). With these techniques, debugging a single core is a relatively well studied problem (still challenging though). However, since embedded cores (processors, DSPs, etc.) have to work together for certain functions, debugging one core at a time can be ineffective and sometimes misleading [18].

For a complex SoC with a number of embedded cores, since they communicate with each other during normal operation, concurrently debugging several cores at a time greatly increase the debug effectiveness and efficiency. In addition, as the on-chip communication scheme is an important part of the design, designers often want to see the transactions between cores to identify the root cause of bugs and/or the system performance bottleneck. As a result, an effective multi-core debug solutions should fulfill the following requirements:

- concurrent debug access to multiple embedded cores;
- inter-core transaction tracing and analysis;
- real-time tracing of debug components;
- cross-triggering among debug components;
- low DfD overhead in terms of area, routing and device pins;

Several multi-core debug solutions were proposed by introducing various on-chip instrumentation (OCI) blocks customized for diverse processors, logic cores and embedded buses (e.g., First Silicon's debug solution [12, 18] and ARM CoreSight [4]). The above multi-core debug solutions for bus-based systems, however, are not readily applicable for NoC-based SoCs. First of all, the drastic change in the communication infrastructure affects the debug access mechanism.In addition, the new inter-core communication scheme (i.e., the NoC) requires new tracing and analysis methods. Moreover, embedded cores often work asynchronously and this makes the distribution of trigger events very difficult.

There are also limited works related to the debug of NoC-based systems. Ciordaş et al. [7] proposed to attach dedicated monitoring probes to the NoC routers, which provides basic observability of the NoC in the form of bits. Later, to increase debug efficiency, the same authors [8] improved their monitoring probe to be able to monitor transactions by reconstructing them from the bit-level monitored data. Although [7,

978-1-4244-1922-7/08/$25.00 ©2008 IEEE

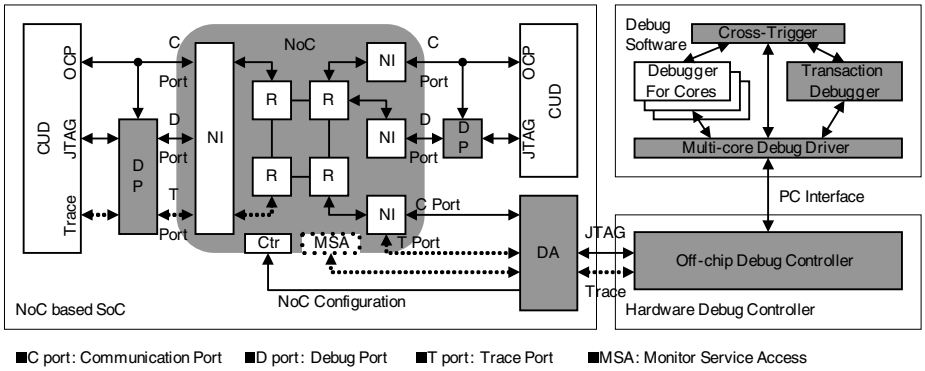


Figure 1: Debug Platform for NoC-based Systems

8] is powerful for finding bugs inside the NoC, it is not suitable for debugging the entire system because an effective debug process 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 debug architecture. In fact, the monitoring service presented in [7, 8] can be treated as built-in DfD logics for the NoC itself, similar to the debug functionalities built into an embedded processor (e.g., ARM in-circuit emulator (ICE) [3]) and logic cores. They can be used to facilitate to precisely pin-point the exact location of the bugs after determining whether the bug exists inside the CUD or inside the NoC. The above observations has motivated us to develop a debug platform suitable for NoC-based systems [19], as briefly introduced in Section 3.

3. NOC DEBUG PLATFORM OVERVIEW

The debug platform in [19], as shown in Fig. 1, is composed of three main parts, *on-chip debug architecture*, including debug probes (focus of this paper) and a debug agent, *off-chip debug controller* and *supporting debug software*. Debug commands and data are transferred through NoC connections with guaranteed quality of service (QoS) in terms of throughput and latency, which is flexible for different debug applications as we can simply adjust the NoC bandwidth used for debug[2]. Using NoC connections for debug access also reduces the routing overhead as no dedicated wires are introduced. For the ease of discussion, we assume that embedded cores communicate with each other using the OCP protocol [15]. Note that, however, the debug platform works with other transaction based communication protocols (e.g., AXI [2] or DTL [16]) as well.

Supporting Debug Software: The supporting debug software provides the graphical user interface (GUI) and/or command line interface for controlling the debug process and displaying the debug results and other information. The supporting debug software is made up of three layers of different tools. In the middle, various core specific debuggers from core providers (e.g., ARM [4]) and a transaction debugger control and observe corresponding debug entities. Upon these debuggers, a cross debugger communicates to them at the same time and controls multi-core cross-debug. 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, such as parallel interface or ethernet. Since multiple debug requests/information may arrive the driver asynchronously, the driver needs to sort them and add/remove addressing labels to/from them.

Off-chip Debug Controller: The off-chip debug controller serves as a translation layer between the software debuggers and the on-chip debug architecture, which builds transparent connections between them. It receives the debug commands or sends debug data from/to software debuggers, schedules them and controls the on-chip debug agent (discussed later) through the system-level JTAG interface. In addition, the off-chip debug controller uses a TDM (Time Division Multiplexing) scheme for sharing single chip-level trace port among multiple CUDs that send out trace data in real-time (detailed in Section 4.3).

On-chip debug architecture: The on-chip debug architecture is the core of the debug platform. It mainly contains the following components:

- a debug probe (DP) between every CUD and its network interface;
- a system-level debug agent (DA) that provides multi-core debug access through a single chip-level JTAG interface and an optional trace port;

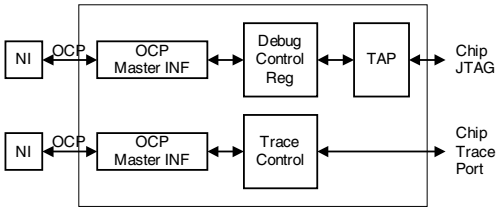


Figure 2: Block Diagram of Debug Agent

As the DP is the focus of this paper, it will be discussed in detail in Section 4. Fig. 2 gives an overview of the debug agent. The DA translates the command sequences on the chip-level JTAG port into read/write operations of on-chip debug resources. With QoS guaranteed NoC channels between the DPs and the DA, we can map all debug resources (e.g., debug control and status registers inside DPs or CUDs) to memory-mapped address space as the slave of the DA and access them in a unified manner. We introduce an extra JTAG instruction "DEBUG_REG" and the corresponding JTAG data register "DEBUG_REG_DATA" for accessing the debug resources 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 can then read or write debug registers inside the chip through standard JTAG interface, as shown in the following table.

[2]We assume that the NoC can be reconfigured through its control interface as in [9].

Off-chip Debug Controller	Debug Agent
Write 'DEBUG_REG' command into IR;	Use DEBUG_REG_DATA 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;

The DA also supports the optional chip-level trace port controlled by the off-chip debug controller, where the software debuggers are able to access the trace data inside the DPs and CUDs in real-time, if any.

4. DEBUG PROBE

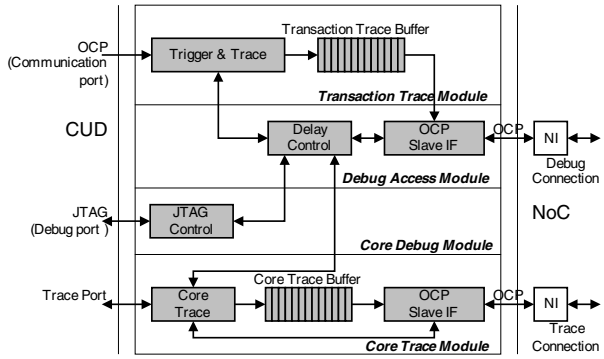


Figure 3: Debug Probe Block Diagram

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 out of the chip for further analysis. On the NoC side, the DP need two NoC connections: one for debug access and the other for the trace (if the trace function is needed).

As shown in Fig. 3, the debug probe is composed of four main components: *transaction trace module*, *debug access module*, *core debug module*, and *core trace module*. Among them, the core trace module can be removed for those CUDs that do not have trace interfaces. In addition, the functionalities of these components can be enhanced or degraded according to the debug requirements(e.g., adding or removing triggers). With this scalable architecture, designers are able to trade-off the cost and the functionalities of the DP. We detail the design of these modules in the following.

4.1 Debug Access Module

The debug access module provides an OCP slave interface as the service access point of the DP. All debug resource are mapped into a unified address space and accessed through memory-mapped OCP read/write operations.

The OCP slave interface decodes the OCP address to distinguish between the registers and the buffer, and then passes the register access to the delay control unit. Concurrent debug of multiple cores usually requires synchronizing the debug operations of multiple DPs and CUDs by 'delaying' these operations properly [19]. This is done by first setting appropriate values to the delay counter in the delay control unit before really issuing debug commands. The delay counter will then count down on its local clock to zero before the debug operation is performed. For example, to write a DP's register after five clock cycles, debugger should set the initial delay value as five before the write operation (so-called "delayed write").

4.2 Core Debug Module

The core debug module controls and observes the CUD through its debug interface (usually JTAG), by translating the debug register read/write commands from the debug access module into the CUD's debug access protocol. For example, our experimental design implements the JTAG debug protocol used for ARM7TDMI core (i.e., embedded ICE [3]). In this module, the access to CUD's debug registers is achieved by certain JTAG sequences. From the external core debugger point of view, these translations are transparent except extra processing delay.

4.3 Core Trace Module

The core trace module 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 connections. However, since there is only one chip-level trace port in most cases, when several CUDs have data to send out at the same time, the "many-to-one" transfer may result in contentions. To avoid that problem, a TDM scheme is used at the chip-level trace port, which is controlled by the off-chip debug controller. By assigning the time slot to each CUD, it grants certain amount of tracing bandwidth(as shown in Fig. 4).

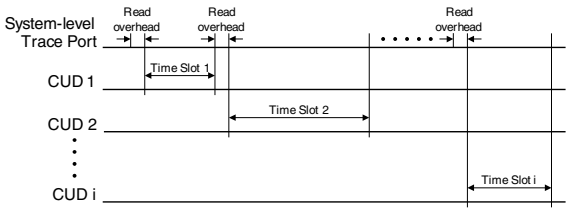


Figure 4: Assign the Time Slot for Each CUD

Apparently, the sum of the bandwidth assigned to the CUDs can not exceed the total bandwidth supported by chip-level trace port. As a result, we may need to limit the trace data generated by the CUD. Given the bandwidth of the chip-level trace port (B_{chip} bps), the number of traced CUDs (N), the time slot for each core $CUD(i)$ ($TS(i)$ $seconds$), and the necessary interval (read overhead, such as set the read address etc.) between two read operations (T_{int} $seconds$), the following equation gives the available bandwidth for $CUD(i)$:

$$B_{CUD(i)} = B_{chip}\left(\frac{TS(i)}{\sum_{i=1}^{N}(TS(i)+T_{int})}\right) \ (bps) \quad (1)$$

Even if the CUD is assigned with enough trace bandwidth, the DP still needs to buffer trace data when waiting for its time slot. For $CUD(i)$, it has to wait for $NT_{int} + \sum_{j\neq i}^{N} TS(j)$ $seconds$. During this time, the volume of trace data $TD(i)$ from $CUD(i)$ is as follow:

$$TD(i) = B_{CUD(i)}(NT_{int} + \sum_{j\neq i}^{N} TS(j))$$
$$= B_{chip}\left(\frac{TS(i)}{\sum_{i=1}^{N}(TS(i)+T_{int})}\right)(NT_{int} + \sum_{j\neq i}^{N} TS(j)) \ (bits) \quad (2)$$

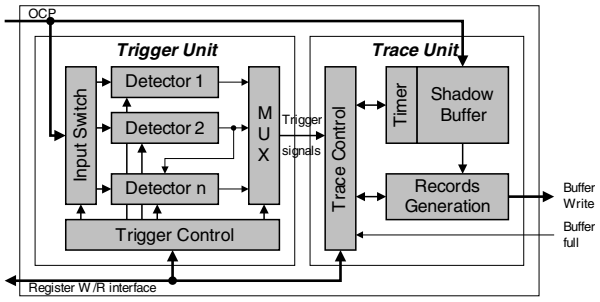


Figure 5: Transaction Trigger and Trace Unit

In case the time slot for every CUD is the same (TS), we get:

$$TD(i) = B_{chip}(\frac{TS}{N(TS+T_{int})})((N-1)TS+NT_{int})$$
$$= B_{chip}(TS-\frac{TS^2}{N(TS+T_{int})})\ (bits) \quad (3)$$

As the T_{int} is generally much smaller than TS, the trace data for $CUD(i)$ can be further simplified as follows:

$$TD(i) \approx B_{chip}(TS(\frac{N-1}{N}))\ (bits) \quad (4)$$

As above equations, larger time slot provides more efficient bandwidth utilization (larger $B_{CUD(i)}$), but requires larger trace buffer because every core has to wait longer for the next time slot.

4.4 Transaction Trace Module

The transaction trace module monitors inter-core transactions and records them based on configurable trigger conditions. The key building block in this module is a transaction trace and trigger unit, as shown in Fig. 5.

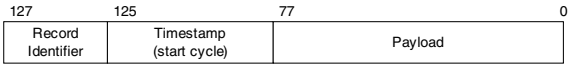
Trigger Unit: The modular trigger unit contains a number of detectors (see Fig. 5) which realize various trigger conditions. The inputs of these detectors are chosen from the OCP signals with an input switch; while their outputs are supplied to a MUX to generate the final trigger signals.

A detector can be a simple two-input comparator, which just compares a set of OCP signals (e.g., command, address or data) to a runtime configurable value. A more complex detector can be a transaction analyzer that is able to identify transaction errors or certain patterns. The basic function of the transaction analyzer in our experimental design is to check whether the transactions conform with the OCP protocol. It detects errors such as 'wrong command coding', 'illegal address', 'no command acceptation', 'no response', and 'wrong response'. It can be extended for different debug requirements and fitted in the system with well defined interface. For example, by embedding some counters, it is able to calculate the delay of a transaction (e.g., the round-trip read delay), for evaluating whether the NoC design achieved the QoS target.

In addition, multiple detectors can work together to form more complex trigger conditions by supplying one detector's output to other detectors as part of its inputs or even as the its enable/disable control signal. As an example, we need a DP to "trigger when the transaction initiator reads address $0x8000xxxx$ but do not get response data in 20 clock cycles". We can combine a comparator that checks the OCP command and address signals and a transaction analyzer which counts the response time. Instantiating how many detectors is a design-time decision, but what the trigger conditions can be run-time controlled by programming them.

Trace Unit: The trace unit is responsible for recording the transactions with predefined format on a trigger event (see Fig. 5). However, as the trigger event may be activated several clock cycles after the transaction message starts, if we only store the transactions at the trigger time, some essential information may be already lost. To tackle this problem, we design a shadow buffer inside the trace unit to store the "start cycle" of the transaction and its timstamp temporarily. Usually, this cycle can be identified from the "master command". In addition, the "trigger cycle" and its timestamp are also stored for generating the transaction records.

When the trigger event is activated, part of the data in the shadow buffer are fetched out to generate a transaction record and then write to the transaction trace buffer by the *transaction records generation unit*. To simplify the record generation process and buffer control mechanism, we use a fixed-length record format as follows:

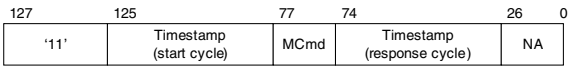


The *record identifier* is a predefined codeword that identifies the trigger event associated to this record. The width of the record identifier depends on the number of the detectors. Each bit indicates the trigger status of one detector. The *timestamp* represents the time when the transaction starts. It is the number of clock cycles passed from the start of the debugging. The *payload* field contains selected transaction data. In most cases, designers only need to see part of the transaction data(e.g., commands and address) in one debug iteration. Therefore, the record generation module supports flexibly choosing part of the OCP signals stored in the shadow buffer by setting a payload mask. Only those signals whose corresponding mask bits are set will be stored in the payload of the record. By doing so, the length of the record payload can be effectively reduced without losing debug accuracy. The size of payload is a design-time trade-off between better observability and smaller area and traffic cost. In our experimental design, we chose 2-bit record identifier (for two detectors), 48-bit timestamp and 78-bit payload to build a 128-bit width record.

For example, designer wants to know the round-trip delay when the CUD reads from address $0x80000010$. Then the "read started start", which is the timestamp of the "start cycle", and the "response time" which is the timestamp of the "trigger cycle" have to be recorded. The "start cycle" is the cycle where the OCP master command (OCP signal MCmd) is $3'b010$ ('read' operation) and the master address (OCP signal MAddr) is $32'h80000010$. The "trigger cycle" is detected when the OCP slave response (OCP signal SRep) is 2'b01 (valid data). So we set the trigger and payload mask as follows:

1. set trigger condition:
$MCmd = 3'b010 \bigwedge MAddr = 32'h80000010 \bigwedge SRep = 2'b01$
2. set payload mask to enable "OCP MCmd" of start cycle;
3. set payload mask to enable "Timestamp" of trigger cycle;
4. start debug;

With the above settings, when the trigger event is activated, the "timestamp" and "MCmd" of "start cycle" and the "timestamp" of "trigger cycle" are fetched from the shadow buffer and then filled in the "timestamp" and "payload" fields of the record respectively as:



The transaction records are firstly written to the transaction trace buffer and then wait for the off-chip debug controller to read it out. A straightforward method is to fetch the entire trace data out at once after the debug process terminates. However, it requires that the transaction trace buffer is large enough to store all the records, whose number

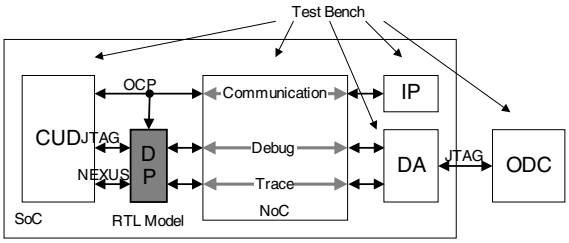


Figure 6: Simulation Environment

is not even predictable. Another strategy is to fetch the debug data constantly during debug process before the buffer is full. But, due to the bandwidth limitation of the debug connections, reading out the trace data in real-time may be quite difficult. Therefore, as a trade-off, we use reasonable buffer size with 'best effort' read policy, i.e., the trace data is read out as soon as when no other high-priority operations (e.g., debug control) uses the same debug connection. Clearly, the transaction buffer may overflow under this strategy. In this case one bit in the debug status register is set to indicate the buffer overflow. We expect designers to set a tighter trigger condition to avoid buffer overflow in the next debug run in case it happens. Since the debug process itself is a "try-and-error" process and usually requires multiple iterations to pinpoint the bug locations, we believe this debug strategy will not increase the debug time significantly.

4.5 DP Programming Model

By default, the DP is disabled and only debug access module is listening to the debug commands, so that the power consumption is reduced in normal functional mode. Under such circumstance, the DP needs to be configured and enabled first before the debug process starts.

Most modules can work with default settings and do not need to be reconfigured frequently except for the *Transaction Trace Module*, which should be programmed in the following sequence in one debug iteration:

1. Select the inputs of the detector and the transaction analyzer from the OCP signals;
2. Set the trigger output of MUX.
3. Set the operation parameters of the detector and the transaction analyzer;
4. Set the start and stop counters;
5. Set the payload mask for record generation;
6. Set the timestamp counter initial value;
7. Set the 'enable' bit in the control register to start the trigger and trace process;

5. EXPERIMENTAL RESULTS

5.1 Functional Simulation

To verify the functionalities of the debug probe described in Section 4, we build a simple yet effective simulation environment, as shown in Fig. 6. The behavioral model of the CUD implements the debug-related functions of a processor, which serves as the transaction initiator that communicates to another IP module (a memory core here) through a NoC channel. During the simulation, the CUD reads and writes the IP core randomly, where these transactions are monitored and recorded by the DP. The debug port is functionally equivalent to the ARM Embedded ICE JTAG port and the trace port is a NEXUS auxiliary port [14] that transports messages containing traced data with predefined throughputs. The off-chip debug controller generates JTAG debug commands for setting up the DP's debug registers as described in Section 4.5. These commands are translated by the DA and then access the debug resources in the DP or the CUD through NoC debug connections. It should be noted that we use connections with pre-defined delays to model the QoS-guaranteed NoC communication channels. We conduct two simulations to illustrate the functionalities of the proposed DP, as discussed in the following paragraphs.

Fig. 7 shows a 'delayed write' operation to the CUD's debug control registers. In the beginning, we set the delay counter register in the "Debug Access Module" with a pre-defined value ('5' in this example). Next, we send the address of the target debug control register inside the CUD and the data to be written into it to the CUD's JTAG interface. As can be observed from the figure, since the value of the delay counter is '5', this write operation is delayed for 5 clock cycles. When the CUD's JTAG interface unit receives the address and data, they are transferred to the CUD (with protocol specified in [3]). Finally, the Debug Control Register is set with expected value after the JTAG sequence is complete. Since the JTAG sequence for a specific register is fixed, we can control the time when the debug commands (by setting debug control register) take into effect with the above method. Similarly, the debug registers inside the DP can also be written with a pre-determined delay.

Fig. 8 presents an example on the transaction trace process. In this experiment, we first program the trigger and trace registers as described in Section 4.5.The DP then monitors the CUD's OCP port continuously according to the pre-assigned configurations. When one of the transactions try to read from the specified address, the detector starts the transaction analyzer (as detailed in Section 4.4). If the transaction analyzer find that the delay of the response is larger than the pre-defined value, it triggers the "Record Generation Unit". Finally, this transaction is recorded after one clock cycle into the transaction trace buffer for further analysis.

5.2 Debug Probe Cost Analysis

As introduced in Section 2, the on-chip debug architecture contains a number of debug probes. Therefore, the area of the proposed 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.

Reference	Area (μm^2)	Percentage
Debug Probe Top	188059.0	100.0%
Transaction Trace Module		
Trigger and Trace Top	19692.1	10.5%
Input Switch	481.8	0.3%
Detector	543.3	0.3%
Transaction Analyzer	364.4	0.2%
MUX	14.3	0.0%
Record Generation	4629.7	2.5%
Shadow Buffer	9197.9	4.9%
Trigger and Trace Control	4456.3	2.4%
Transaction Trace Buffer	120689.9	64.2%
Debug Access Module		
Delay Control	5384.8	2.9%
OCP Slave INF	2020.7	1.1%
Core Debug Module		
JTAG INF	3912.9	2.1%
Core Trace Module		
Core Trace INF	3099.6	1.6%
Core Trace Buffer	31225.6	16.6%
OCP Slave INF	2024.0	1.1%

Table 1: DfD Area Cost of the Debug Probe

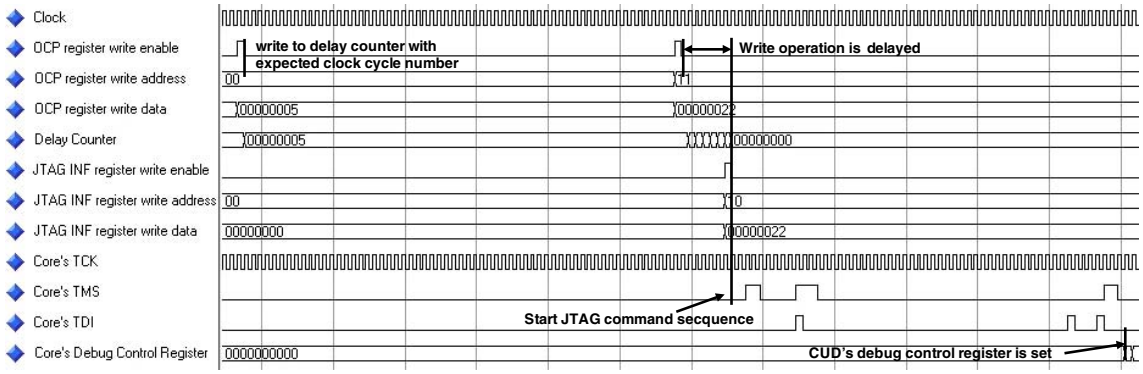


Figure 7: Simulation Waveform for the Delayed Write Operation

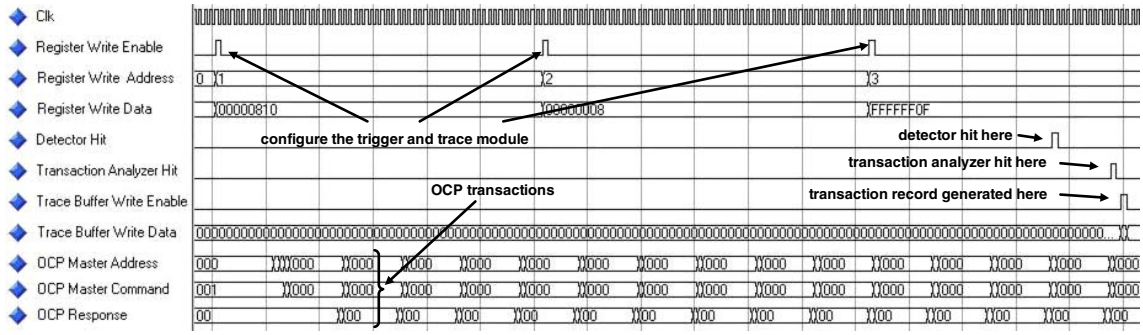


Figure 8: Simulation Waveform for the Transaction Trace Operation

Table 1 presents the detailed area data of all the blocks shown in Fig. 3. 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 instead of registers to form these buffers, their areas 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 control logic significantly.

The area of the transaction trace buffer depends on the number and width of the trace records, which differs from one application to another. The core trace buffer size varies with different cores' debug requirements and also the time slot sizes as analyzed in Section 4.3. Generally speaking, our DP design provides the flexibilities for designers to make the best trade-off according to their debug needs.

6. CONCLUSION

In this paper, we present the detailed design and analysis of a novel debug probe for the NoC-based systems, which is composed of four main components: transaction trace module, debug access module, core debug module, and core trace module. The DP is well fitted in the debug platform presented in [19] to facilitate concurrent debug of embedded cores and inter-core transactions, at a moderate DfD area overhead.

7. REFERENCES

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

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

[3] ARM Ltd. *The ARM7TDMI Debug Architecture*, Dec. 1995. ARM DAI 0028A, 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, January 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, T. Basten, A. Rădulescu, K. Goossens, and J. van Meerbergen. An event-based monitoring service for networks on chip. *ACM Transactions on Design Automation of Electronic Systems*, 10(4):702–723, Oct. 2005.

[8] C. Ciordaş, K. Goossens, T. Basten, A. Rădulescu, and A. Boon. Transaction Monitoring in Networks on Chip: The On-Chip Run-Time Perspective. In *Proc. IES*, 2006.

[9] 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, Sept-Oct 2005.

[10] K. Goossens, B. Vermeulen, R. van Steeden, and M. Bennebroek. Transaction-Based Communication-Centric Debug. In *Proc. NOCS*, pp. 95-106, 2007.

[11] 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, July 2006.

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

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

[14] The NEXUS 5001 Forum. http://www.nexus5001.org.

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

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

[17] The International Technology Roadmap for Semiconductors (ITRS): 2003 Edition.

[18] N. Stollon, R. Leatherman, B. Ableidinger, and E. Edgar. Multi-Core Embedded Debug for Structured ASIC Systems. In *Proc. DesignCon*, 2004.

[19] S. Tang and Q. Xu. A Multi-Core Debug Platform for NoC-Based Systems. In *Proc. DATE*, pp. 870-875, 2007.

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

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

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