In-band Cross-Trigger Event Transmission for Transaction-Based Debug

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

Cross-trigger, the mechanism to trigger activities in one debug entity from debug events happened in another debug entity, is a very useful technique for debugging applications involving multiple embedded cores. Existing solutions rely on dedicated interconnects (i.e., different from functional interconnects) to transfer debug events and cannot guarantee the arrival time of the debug events coincides with the arrival time of the data messages between multiple cores. This results in mismatches between the observed system internal operations and the ones that designers expect to watch. To tackle the above problem, in this paper, we propose to package the cross-trigger events and the actual data together into transaction messages and transfer them along the same functional interconnects (namely in-band debug event transmission), with the help of novel design-for-debug circuits. Simulation results on a hypothetical NoC-based systems show the effectiveness of the proposed technique[1].

1 Introduction

With the ever advancement of the semiconductor technology, today's high-performance system-on-a-chip (SoC) devices are able to integrate hundreds of millions of transistors on a single silicon die. To reduce time-to-market, SoC designs are usually created by combining many pre-designed intellectual property (IP) cores (e.g., processors, DSPs and memories) and custom user-defined logic (UDL). These embedded cores typically communicate with each other via functional interconnects such as on-chip buses or network-on-chip (NoC) [5, 8].

With more functionalities integrated onto the chip, it is necessary to improve the designs' debug support at a pace proportionally to the increasing design complexity [10]. While the traditional postmortem debug technique that captures snapshots of the system through JTAG run-control interface is still the most widely-utilized method [18], various research groups have proposed to embed more design-for-debug (DfD) structures on-chip for hardware tracing difficult-to-find bugs (e.g., [1, 3, 4, 11, 12, 19]). With the above techniques, debugging a single embedded core is a relatively well studied problem (still challenging though). For SoCs running applications that involve multiple embedded processors and/or other IP cores, however, debugging one core at a time can be sometimes misleading because a bug may not exhibit itself until all related parties are controlled and observed at the same time. Multi-core debug solutions (e.g., [2, 15]) are essential to correctly analyze the system's internal operations in such cases.

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

In multi-core debug architecture, one of the key design factors is how to trigger a particular operation in one debug entity (e.g., an IP core) from a debug event happened in another debug entity. This so-called cross-trigger capability is essential when debugging complex interactions among multiple cores. For example, consider in an SoC device for a 3G cell phone, two embedded cores perform digital signal processing (DSP) and protocol tasks, respectively. They communicate with each other in the form of transactions. Suppose, in one transaction, we observe an erroneous response message from the DSP core to the protocol core. This error can result from synchronization problems during core interaction, computational errors inside the DSP core, or even bugs inside the protocol core. For a better understanding of this error, it would be really helpful if we are able to stop the DSP core's operations and observe its internal states, triggered by the transaction initialized at the protocol core.

Cross-trigger typically involves three parties: the trigger source that generates trigger event when certain conditions are fulfilled; the trigger target that performs debug actions and trigger event delivery mechanism that transfers the trigger event from the trigger source to the trigger target. It can be observed from the above example that, in many cases, cross-trigger are related to certain inter-core transactions. We name such trigger events as *transaction-related cross-trigger events (TRCT-events)* in this paper.

Existing multi-core debug solutions (e.g., [2, 9, 15]) utilize dedicated interconnects (i.e., different from functional interconnects) to distribute debug events to implement cross-triggering. While this technique has the benefit to be able to transfer any debug events at any time, it has a main limitation when delivering TRCT-events: the arrival time of the cross-trigger event and the related transaction message are often misaligned because they are transferred through different paths. If the cross-trigger event arrives early, the target entity may be stopped at an unexpected state and require cumbersome debug control to walk into the expected state. If the cross-trigger event arrives late, the target entity may have already passed the expected state and the information that we want to observe is lost.

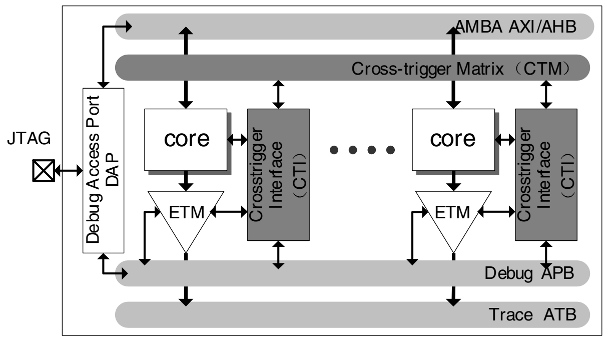
AMBA AXI/AHB
Cross-trigger Matrix（CTM）
JTAG
Debug Access Port DAP
core
Crosstrigger Interface（CTI）
ETM
core
Crosstrigger Interface（CTI）
ETM
Debug APB
Trace ATB

Figure 1. ARM Cross-Trigger Infrastructure [2]

As the timing of the debug event transmission plays a vital role for the effectiveness of the cross-triggering mechanism, in this paper, we propose to transfer the transaction-related cross-trigger events along with the actual associated transaction data via the same functional interconnects. We call this strategy *in-band cross-trigger event transmission* and we introduce novel core-level DfD circuits to implement it. By doing so, we guarantee the timing of the trigger event coincides with the actual transaction message and hence we are able to control and analyze the system's internal operations precisely. A case study on a NoC-based system shows the effectiveness of the proposed technique. We have also demonstrated the overhead and limitations of the proposed approach.

The remainder of this paper is organized as follows. Section 2 reviews the related work in this domain and motivates this work. The proposed in-band cross-trigger event transmission strategy and its implementation details are described in Section 3 and Section 4, respectively. Next, Section 5 presents experimental results for a hypothetical NoC-based system. Finally, Section 6 concludes this paper.

2 Prior Work and Motivation

The key implementation issue in a cross-triggering mechanism is how the trigger events are distributed in the system. A typical solution is presented in the ARM CoreSight debug architecture [2], where a novel *embedded cross trigger (ECT)* unit, consisting of a number of *cross-trigger interface (CTI)* blocks and a *cross-trigger matrix (CTM)*, is specified to pass the trigger event from one embedded core to another. As shown in Fig. 1, the CTI combines and maps the trigger requests from an embedded core and/or its associated *embedded trace module (ETM)*, and broadcasts them to all the CTI blocks in the ECT as channel events or re-maps these events onto a trigger output after receiving them. Dedicated interconnects are employed in the CTM to connect multiple CTIs.

While the above solution focused on debugging systems from a computation-centric point of view, transaction-based debug strategies have been advocated by several research groups with the increasingly importance of communication in today's electronic systems [9, 16].The authors in [16] proposed a system-level debug solution for NoC-based systems and introduced a debug probe to generate trigger events based on predefined configurable trigger conditions. It is mentioned that these trigger events are transferred by reusing the NoC links, but no details are presented. Goossens *et al.* systematically discussed communication-centric debug in [9] and introduced dedicated *event distribution interconnect (EDI)* that "deliver events from the monitors and other event generators, such as the TAP controller, to the relevant debug components, such as the network interfaces and TAP controller". Debug events can be quickly distributed on their proposed EDI as they are broadcast synchronously at high functional frequency by pipelined stop module.

To show the limitations of the current cross-triggering mechanisms, a typical multi-core debug scenario is presented in Fig. 2. Core B runs a complex computation task and updates its status registers (0x0004) from time to time. Core A is running a complex state machine to control the operation of the whole chip. In some states, Core A needs to check Core B's status by reading address 0xF0000004 ('F' is mapped to a remote read transaction to Core B.). Suppose we encounter an unexpected state transition after checking Core B's status, which indicates an error happens on the transaction path that involves hardware/software of core A, core B and the interconnects between them. Debugging a single core at a time is not good enough to identify this type of bug as it does not have the knowledge of the complete context of the error. We need cross-trigger mechanism to discover the root cause of this problem.

In computation-centric debug solutions such as [2], a breakpoint can be set before or after the corresponding "remote load" instruction in core A to stop its execution and cross-trigger a 'stop' operation at core B. In the first case, the read transaction has not even executed and hence it is impossible to identify the error. If the breakpoint is set after 'load', since the read transaction has already been completed at this moment, the chances to observe intermediate operations in core B and the interconnects will disappear. Transaction-based debug solutions (e.g., [9, 16]) are able to link the trigger event with the actual transaction by generating it according to the signals on core A's communication interface. Unfortunately, as the cross-trigger event goes through a path different from the functional transactions, the trigger event may arrive core B earlier or later than the expected time (i.e., the time when the actual transaction arrives core B), which causes core B to stop at an unexpected state. Although [9] is able to block further transactions coming out of core A, the content in address 0x0004 can be a misleading value written at a different time inside core B. Tracing before and/or after the trigger event happening is helpful for tolerating the timing uncertainty, but it requires large trace buffer and also cannot guarantee to solve the problem.

In above debug scenario, when debugging inter-core transactions, ideally, we should stop and observe the master core's behavior from the point where it issues the transaction, monitor the interconnects from the point that the transaction transfers along them, and stop and observe the slave core exactly at the time that the transaction request arrives. This, however, cannot be easily achieved with current debug solutions. The

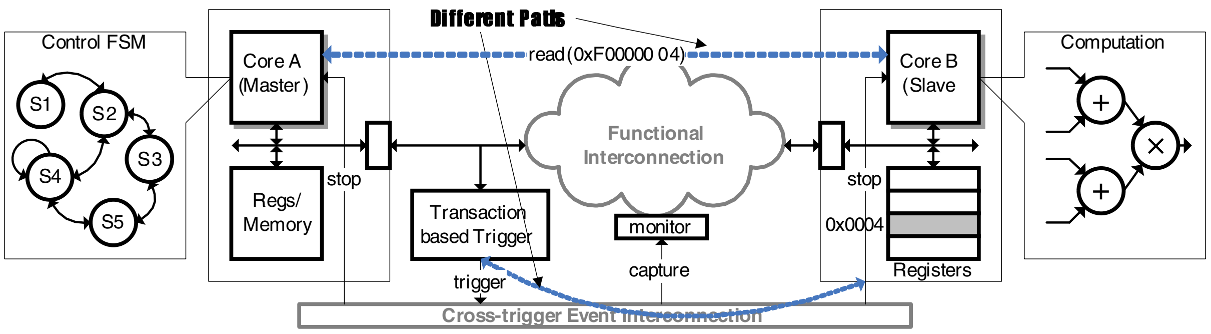

Figure 2. Different Paths for Transaction Messages and Trigger Events

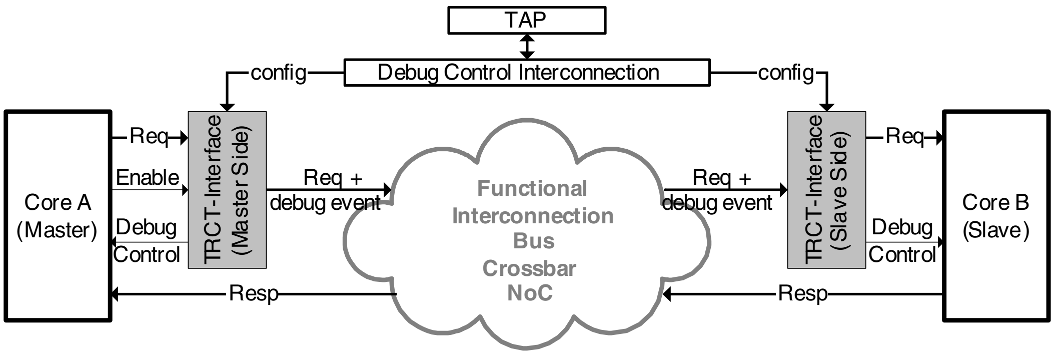
TAP
config
Debug Control Interconnection
config
Core A (Master)
Req
Enable
Debug Control
Resp
TRCT-Interface (Master Side)
Req + debug event
Functional Interconnection Bus Crossbar NoC
Req + debug event
TRCT-Interface (Slave Side)
Req
Debug Control
Resp
Core B (Slave)

Figure 3. In-band Debug Event Distribution

above observations motivate us to investigate a better way to generate and distribute TRCT-events to achieve more effective cross-triggering. Our solution is simple yet effective: we try to *generate and transmit the trigger event together with the corresponding transactions along the same path*, detailed in the following sections.

3 In-band Trigger Event Transmission

We borrow the word 'in-band' from the telecommunication literature, where the in-band signaling and the voice data are transmitted along the same communication channel. Despite the underlying media and the length of the communication path, the signaling and the corresponding data arrive the destination at the same time. We apply the similar concept to the transmission of the cross-trigger events and its related inter-core transactions, and we name it *in-band trigger event transmission*.

As demonstrated in Fig. 3, core A and core B are the master and the slave of a particular transaction, connected via functional interconnects such as bus, cross bar, or NoC. New transaction-related cross-trigger interfaces (TRCT-interfaces) are introduced between the core's communication ports and the functional interconnects for both the master core and the slave core. When core A starts a transaction by sending out a request message, master side TRCT-interface analyzes the message and generates a trigger event when predefined trigger conditions are qualified. The outgoing message of the TRCT-interface is a combination of debug event (coded signals as discussed later) and the original request. This 'Req + trigger event' message is transmitted via the underlying interconnects and arrives core B's slave TRCT-interface, where the original request is directed to. The debug event are then used to generate appropriate debug control signals (e.g., stop signal for core B) for its execution. In addition, we can equip other debug entities (e.g., the bus monitors, the NoC's network interfaces or routers) with capabilities to 'recognize' the in-band trigger events, so that they can use them as the 'signaling' to trigger certain debug actions. If these entities are not provided with such capabilities, the cross-trigger events are simply treated as part of the functional transactions and bypassed.

With the proposed in-band trigger event transmission mechanism, no matter how the on-chip interconnection are implemented, the cross-trigger event reaches the target together with the corresponding transaction. Therefore, the problems described in Section 2 can be solved, i.e., we can stop core B and check the content of the memory address 0004 just at the time when the transaction request arrives it.

Similarly, the above concept can be used for the response messages, in which the only difference is that the trigger events are generated at the slave side and transmitted to the master side.

4 Implementation in NoC-Based Systems

For a simple single-layer bus-based system, as the delay of the transaction messages and the delay of the trigger events are usually small, typically their arrival time difference is also small when they are transferred along different paths. Therefore, this problem is relatively easy to deal with by introducing trace buffers with acceptable size. However, in a large-scale SoC with more complex on-chip communication

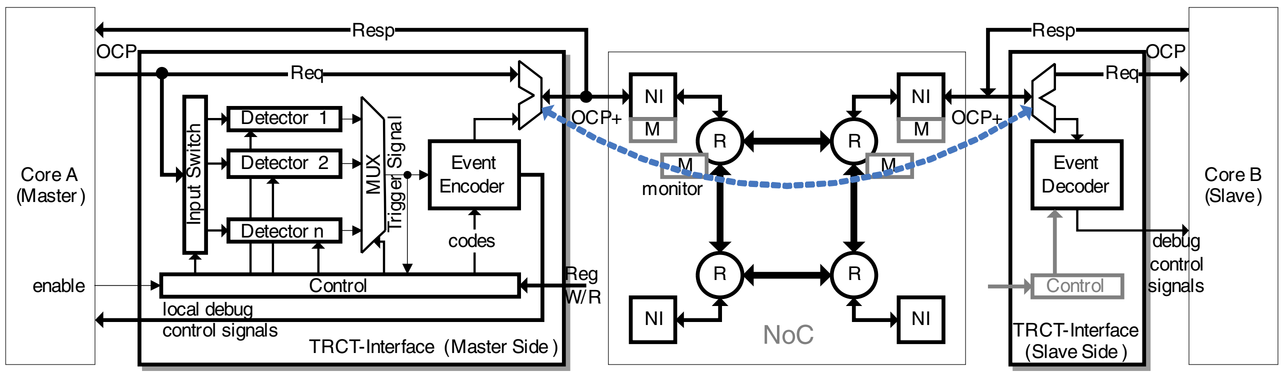

Figure 4. Implementation in NoC-Based Systems

schemes such as multi-layer bus or NoC, the difference between the arrival time of the trigger events and the one of the corresponding transactions could be quite large and unpredictable. For example, in [6], the average packet latency is shown to be inversely dependent on the bandwidth guarantees, typically around 30 to 50 clock cycles. Latency overhead introduced by the NI (Network Interface, refer to Sect. 4.1) itself is between 4 and 10 cycles in [14]. Aligning the delay of the TRCT-events and their associated transactions in such systems is therefore quite difficult when they are transferred using different paths. In addition, the SoC may be a globally-asynchronous locally-synchronous (GALS) system, in which the cores and the communication infrastructure are running with different clocks. This makes the delay aligning problem even more complex. The proposed in-band trigger event transmission strategy is more valuable for such systems.

4.1 NoC-Based System Model

In our NoC-based system, an IP core is modeled as a module that has both a communication port and a debug control interface. The communication port is assumed to be compliant with the open core protocol (OCP) [13], which defines a transaction-level interface commonly used in NoC-based systems. To simplify the discussion, the debug control interface is defined as a set of control signals. Enable/disable each of them results in a supported IP core debug operation, such as ‘stop’, ‘step’, ‘resume’ and ‘start trace’. A more complex debug interfaces generally provides similar functions with a different protocol. For IP cores that do not have a debug interface, debug wrappers are required to be implemented to provide a similar debug control interface.

The NoC is assumed to be composed of 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 to provide the raw bandwidth.

4.2 TRCT-Interface Implementation

As discussed above, the master side TRCT-Interface analyzes the transactions and generates coded trigger events for local and remote debug entities; while the slave side TRCT-Interface decodes the debug events to generate the debug control signals. We explain the details in this section.

Transaction Analysis: This is conducted by a number of detectors that realize various configurable trigger conditions (as shown in Fig. 4). 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 compares a set of OCP signals (e.g., command, address or data) to a run-time configurable value. A more complex detector can be a transaction analyzer with counter, sequencer or even finite-state machine (FSM), which is able to identify transaction errors or certain patterns. It can be extended for different debug requirements and fitted in the system with well-defined interface. 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 its enable/disable control signal. Instantiating how many detectors is a design-time decision, but the trigger conditions can be run-time controlled by programming the debug control register inside the TRCT-Interface.

Trigger Events Generation and Coding: The trigger events are generated when the transaction analyzer detects that the trigger condition is fulfilled, and the enable signal from the core is set at the same time. With the enable signal from the core, more precise control of trigger event generation can be achieved. In the example shown in Sec. 2, the program running on core A may read the same address of core B several times, these read requests appear to be the same at the OCP interface and hence the transaction analyzer itself cannot distinguish them. To precisely locate a particular read request, we need to know the exact instruction that initiates the read transaction. This can be done by utilizing the breakpoint/watchpoint of core A to generate the ‘enable’ signal and “tell” the TRCT-Interface when to start the trigger process.

Encoding the TRCT-events reduces the bandwidth requirement to transfer them. In the master side TRCT-Interface, the trigger signals are encoded by pre-defined codewords. For example, with a 2-bit code, we can translate the trigger signal into up to four different events. These coded events are decoded into control signals at the slave side TRCT-Interface and/or by the debug entities along the transaction path. In one debug iteration, different debug entities may translate these events into different actions such as 'stop', 'step', 'resume', 'start tracing' and 'flushing the trace data'.

Control and Configuration of TRCT-Interfaces: The TRCT-Interfaces are controlled by setting its debug control registers through a register write/read interface. There are also some status registers implemented inside these interfaces, e.g., 'trigger happened indication' in the master side TRCT-Interface and 'trigger event received indication' in the slave side TRCT-Interface. In addition, to reduce power consumption in normal functional mode, the whole TRCT-Interface can be disabled by setting the 'stop' bit of the control register.

Generally speaking, for one debug iteration, this module should be programmed in the following sequence:

1. Select the inputs of the detectors 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 codeword for event coding and decoding.
5. Start the trigger process;

Note: As its function is simple, slave side TRCT-Interface may use fixed settings.

Delay Introduced in the TRCT-Interface: The master side TRCT-Interface needs to delay the functional transactions (typically one OCP clock cycle) to align them with the corresponding trigger event. This is generally acceptable in a SoC with complex interconnection infrastructure as the master IP usually does not expect delay-free communication. As discussed earlier, in NoC-based systems, the delay introduced by the NoC itself can be in the range of tens of clock cycles. For cores that are able to tolerate such delays, the extra delay introduced in the TRCT-Interface will not cause unacceptable performance degradation. For the slave side TRCT-Interface, as it just needs 'listening', only a small combinational logic delay is introduced.

Finally, in terms of implementation, the TRCT-Interface can be a stand-alone module or embedded in a more complex DfD component, such as the debug probe proposed in [16, 17]. It should also be noted that a TRCT-interface can contain both master-side and slave-side functionalities.

4.3 Trigger Event Transportation in NoC

TRCT-events are transmitted along the functional transaction path. In a NoC-based system, the path starts from the master communication port (i.e., the inputs to the NI), ends at the corresponding slave communication port (i.e., the outputs of the NI), via the NoC links. There are two ways to transport the extra trigger events together with the functional message.

One way is to expand the OCP protocol and re-implement the NoC to support the extra debug event signals. For example, in addition to current OCP signals, one set of debug event signals can be instantiated similar to the 'side-band' signals in OCP protocol, and the NoC should support the transfer of these signals. Although this is the ideal way to implement the 'in-band' trigger event transmission concept, it is not applicable until changes are made in the communication protocols and the NoC implementations.

An alternative method is to add some signals as part of the functional signals, for example, adding two more signals as additional OCP address bits. Suppose initially the address is 16-bit, the output of TRCT-Interface becomes OCP-compliant port with 18-bit address. From the NoC's point of view, the only difference is that the master core transfers data with 18-bit address instead of 16-bit one. To support this, we can simply re-configure the NI to support 18-bit address (This is easy to achieve as most NI implementation is configurable). In addition, we also need to update the routing address to make correct routing decision. That is, for example, if the original design use the four most significant bits (address[15..12]) to locate the different slave cores while the address bits [11..0] are used for the memory element inside it. When two more bits are added for debug purpose as address [17..16], the NoC should still use address[15..12] to find the slave core instead of the current four most significant bits ([17..14]). Otherwise, wrong routing decision will be made. Since address mapping is typically a basic feature of the NoC, this is applicable in most case. Even for NoCs that do not support flexible address mapping, a small modification of NI can solve the problem.

One important question is what if the NoC itself contains bugs and cannot even deliver the trigger events to the slave core correctly. If this happens, in most cases, there should be some clues for us to narrow down the root cause into the NoC. For example, we can compare the 'trigger event generated' bit in the master side TRCT-Interface's status register and 'trigger event received' bit in the slave side TRCT-Interface to observe whether the transmitted message is received successfully. For the bugs inside NoC, its own DfD modules (such as the debug monitors in [7]) can be employed for further analysis.

Using the above method, the NI needs to support additional debug coded signals, e.g., the two more address bits on the interface for the above example. This results in very small increase of area cost . Another cost is the extra NoC bandwidth required to transmit these signals. While it is hard to give a quantitative analysis because this depends on the NoC implementations, it should be comparable to the bandwidth consumed in the dedicated debug interconnects in prior work, but at a much lower routing cost.

5 Experimental Results

To verify the proposed concept of 'in-band' trigger event transmission, we implement an experimental design as described in Sec. 4.2.

Fig. 5 depicts the simulation waveform for this design. The master side TRCT-Interface listens to core A's OCP interface (the MCmd and MAddr signals, refer to [13]), and triggers an TRCT-event when core A tries to read the register (0x0004) inside core B. As can be observed in the waveform, however, there are two similar transactions: both are 'read on 0xF0000004'. To distinguish them, as discussed earlier, the

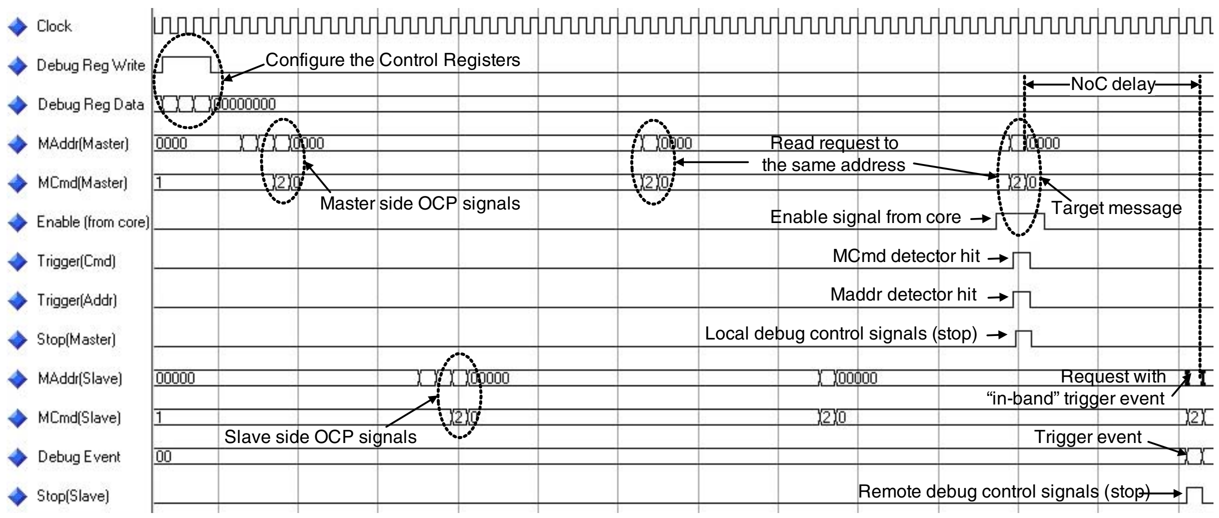

Figure 5. Simulation Waveform for the Experimental Design

'Enable' signal generated by a breakpoint/watchpoint hit inside Core A is used. Next, if the 'Enable' signal is set, the trigger event is coded and sent to core B together with the original read transaction. When the transaction (including the trigger event) arrives core B, the slave side TRCT-Interface decodes the event and stops core B just on time to examine whether the value of register (0x0004) is the expected one. It should be noted that the debug control registers are set as described in Sec. 4.2 before the debug process starts.

As the TRCT-Interfaces need to be inserted in any transaction path where cross-triggering capabiity is required, we need to instantiate a number of them in an SoC. To observe the area cost of the TRCT-interfaces, we implement the above TRCT-Interface with a 0.13 μm commercial CMOS technology. The master side TRCT-Interface is composed of two detectors analyzing the OCP 'MCmd' and 'MAddr' signals and an encoder to generate the coded trigger events. Its total area is 5950 μm^2 (about 1478 equivalent two-input NAND gates). For the slave side TRCT-Interface, it just contains a decoder and a control interface, and its area cost is less than 100 gates. When compared to today's large design size, in general, the area cost to implement the proposed 'in-band' trigger event transmission concept is acceptable.

6 Conclusion

Cross-trigger is a very useful technique for debugging applications involving multiple embedded cores. Existing solutions rely on dedicated interconnects to transfer debug events and hence may result in mismatches between the observed system internal operations and the one that designers expect to watch. To tackle the above problem, in this paper, we propose the so-called in-band debug event transmission concept, in which we package the transaction-related cross-trigger events and the actual data together into transaction messages and transfer them along the same functional interconnects. Experimental results on a hypothetical NoC-based systems prove the effectiveness of the proposed technique at a low design-for-debug cost.

References

[1] M. Abramovici, *et al.* A Reconfigurable Design-for-Debug Infrastructure for SoCs. In *Proceedings ACM/IEEE Design Automation Conference (DAC)*, pages 7–12, 2006.
[2] Advanced RISC Machines Ltd. *CoreSight Architecture Specification*, 2004. ARM IHI 0029B.
[3] Altera Inc. Design Debugging Using the SignalTap II Embedded Logic Analyzer. http://www.altera.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] D. Bertozzi and L. Benini. Xpipes: A network-on-chip architecture for gigascale systems-on-chip. *IEEE Circuits and Systems Magazine*, 4(2):18–31, 2004.
[7] C. Ciordaş, *et al.* Transaction Monitoring in Networks on Chip: The On-Chip Run-Time Perspective. In *Proc. IES*, Oct. 2006.
[8] W. J. Dally and B. Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. In *Proc. DAC*, pages 18–22, 2001.
[9] K. Goossens, B. Vermeulen, R. van Steeden, and M. Bennebroek. Transaction-based communication-centric debug. In *Proc. NOCS*, pages 95–106, 2007.
[10] 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.
[11] R. Leatherman and N. Stollon. An Embedded Debugging Architecture for SOCs. *IEEE Potentials*, 24(1):12–16, Feb-Mar 2005.
[12] MIPS Technologies Inc. EJTAG Trace Control Block Specification. http://www.mips.com.
[13] OCP International Partnership. Open Core Protocol Specification. http://www.ocpip.org.
[14] A. Rădulescu, *et al.* An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming. *IEEE Transactions on Computer-Aided Design*, 24(1):4–17, Jan. 2005.
[15] N. Stollon, R. Leatherman, B. Ableidinger, and E. Edgar. Multi-Core Embedded Debug for Structured ASIC Systems. In *Proc. DesignCon*, 2004.
[16] S. Tang and Q. Xu. A Multi-Core Debug Platform for NoC-Based Systems. In *Proc. DATE*, pages 870–875, 2007.
[17] S. Tang and Q. Xu. A Debug Probe for Concurrently Debugging Multiple Embedded Cores and Inter-Core Transactions in NoC-Based Systems. In *Proc. ASP-DAC*, accepted, 2008.
[18] B. Vermeulen, T. Waayers, and S. K. Goel. Core-Based Scan Architecture for Silicon Debug. In *Proc. ITC*, pages 638–647, Oct. 2002.
[19] Xilinx Inc. Chipscope Pro Software and Cores User Guide. http://www.xilinx.com.