A Breakpoint-Based Silicon Debug Technique with Cycle-Granularity for Handshake-Based SoC

Hsin-Chen Chen[1], Cheng-Rong Wu[1], Katherine Shu-Min Li[2]*, Kuen-Jong Lee[1]**
[1]Dept. of Electrical Engineering, National Cheng Kung University, Tainan, Taiwan
[2]Dept. of Computer Science and Engineering, National Sun Yat-Sen University, Kaohsiung, Taiwan
*smli@cse.nsysu.edu.tw **kjlee@mail.ncku.edu.tw

***Abstract* - The breakpoint-based silicon debug approach allows users to stop the normal (system) operations of the circuits under debug (CUDs), extract the internal states of the CUDs for examination, and then resume the normal operations for further debugging. However, most previous work on this approach adopts the transaction-level or handshake-level of granularity, i.e., the CUDs can be stopped only when a transaction or a handshake operation is completed. The granulations at these levels are often too coarse when a transaction or a handshake operation requires a large number of cycles to complete. In this paper, we present a novel debug mechanism, called the *Protocol Agency Mechanism* (PAM), which allows the breakpoint-based debug technique to be applied at the cycle- level granularity. The PAM can deal with transaction invalidation as well as protocol violation that may occur when a system is stopped and resumed. Experimental results show that the area overhead of the PAM is quite small and the performance impact on the system is negligible.**

I. Introduction

Design-for-debug (DfD) techniques have been proposed to address the debug problem for system-on-a-chip (SoC) [1]. These techniques can be divided into two categories: real-time trace and run-stop-resume debug [1]. The former [2-4] uses trace buffers to record the states of a system thereby providing a way to observe the status of selected signals without interfering with the normal operation of the circuits under debug (CUDs). However, it suffers from limited observable locations and limited size of trace buffers. The latter, also known as breakpoint-based debug (BPD) [5-10], allows users to set breakpoints to stop system operations, dump the internal states of CUDs for examination and then resume the system for further debugging. The BPD provides more debug flexibility and utilizes existing testable design such as scan chains to reduce the hardware overhead. In this paper we focus on the BPD technique.

Previous BPD methods use transaction, message, and/or handshake granularities [5-9]. In such granularities, the current transaction or handshake process must be completed before interrupting a CUD. This may require long delay before one can stop the operations of the CUD. In addition, an IP must first become the bus master before it can access the bus. An IP with a lower priority thus may have to wait for a long time before the bus is granted to it. During this wait time, the circuit cannot be stopped as the acknowledge signal of the handshake process may not be issued. Therefore even if the handshake granularity is adopted, there may still be a long period of time during which the CUD cannot be stopped and the information in the CUD cannot be accessed.

In [10] a BPD platform is proposed for bus-based SoC designs containing embedded processors, embedded memory, IP cores, system bus and bus arbiter. A novel *Test Access Mechanism Controller* (TAM Controller) is developed to carry out the BPD process. The TAM Controller implements the BPD at the cycle-granularity by gating the clocks of all components in the SoC so as to facilitate cycle-based interruption. This clock gating technique may suffer from the fact that some SoC components such as dynamic logic or DRAM cannot be gated. In this work we propose a novel cycle-based debug mechanism called the ***Protocol Agency Mechanism (PAM)*** by which the CUD can be stopped even if there exist IPs whose clocks cannot be gated.

We shall only control the clocks of the circuits under debug. The main problem here is that setting debug granularity at the clock cycle level may break the undergoing handshake, message and transaction, leading to broken transactions. As the handshake targets may not be controlled by the debug controller at the interrupted cycle, unexpected interaction results or even system crash may occur. We shall classify the problems here into two categories: *protocol violation* and *transaction invalidation*. Protocol violation occurs when the interruption of a CUD causes violation of the protocol rules, while transaction invalidation occurs when some handshake targets accept incorrect data or control signals. In this paper we will address these two problems.

II. Overview of SoC with Protocol Agency Mechanism

Fig.1 gives an overview of the hardware architecture of a bus-based SoC system equipped with the proposed Protocol Agency Mechanism (PAM). This SoC contains four types of components: circuits-under-debug (CUDs), intellectual properties (IPs), the bus (interconnect), and a debug controller called the Test Access Mechanism (TAM) Controller [10]. Note that the differences between CUDs and IPs are that CUDs are under debug but IPs are not, which implies that the clocks of CUDs are controllable by the debug controller, while the clocks of IPs are not. Each CUD or IP can have a master or slave port, or both. The PAM consists of 5 types of *Protocol Agents* as shown in the green area of Fig. 1:

1. Protocol agent for the master port of CUD (*CUD_MA*),
2. Protocol agent for the slave port of CUD (*CUD_SA*),
3. Protocol agent for the master port of IP (*IP_MA*),
4. Protocol agent for the slave port of IP (*IP_SA*), and
5. Protocol agent for the bus (*BUS_A*).

978-3-9815370-4-8/DATE15/©2015 EDAA

The CUD_MA (IP_MA) is located between the master port of the corresponding CUD (IP) and the bus. Similarly the CUD_SA (IP_SA) is located at a corresponding slave port. The BUS_A is located in the arbiter of the interconnect. The CUD_MA/SA are controlled by the TAM Controller which works as a breakpoint-based debug controller. The TAM Controller notifies the CUD's Protocol Agent(s) the ascertaining of a breakpoint by signaling "Breakpoint" to all Protocol Agents, while the Protocol Agent of each CUD is responsible for interrupting and resuming the CUD's clock by signaling "Clock Control." When a CUD is interrupted, the corresponding CUD_MA and/or CUD_SA will eliminate transaction invalidation by keeping data consistent between the CUD and its transaction partner. The IP_SA (IP_MA) and the BUS_A will be notified of the interruption by the CUD_MA/SA and they will prevent the IP from issuing new transactions. As a result, both transaction invalidation and protocol violation can be avoided.

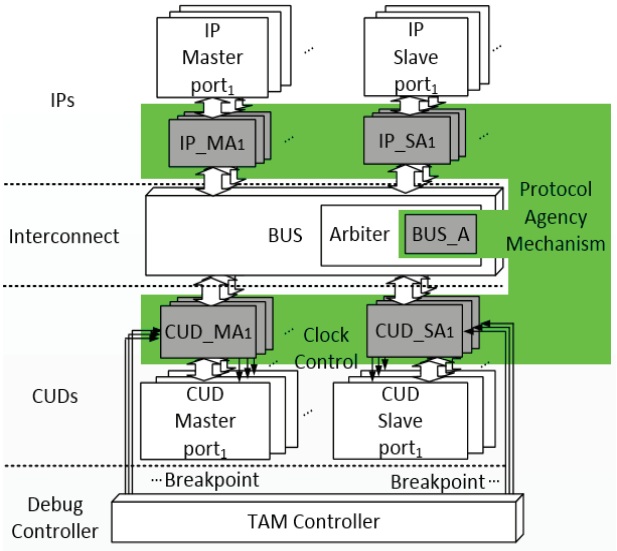

Fig.1. BUS-Based SoC with *Protocol Agency Mechanism*.

III. Operation Schemes in Protocol Agency Mechanism

The PAM consists of two parts: 1) Protocol Agency Mechanism for CUDs (PAM_CUD), and 2) Protocol Agency Mechanism for IP and BUS (PAM_IP_BUS). The goal of PAM_CUD is to ensure data consistency before and after interruption and to prevent the CUD from producing protocol violation, while the PAM_IP_BUS is to hold the proceeding transactions or to mask new transactions during interruption.

The necessity of PAM_CUD is clear as one has to ensure data consistency and protocol compliance when a CUD is interrupted. The necessity of PAM_IP_BUS is explained as follows. Even if a broken transaction does not occur during the debug procedure, other system components may access the interrupted CUD. If another IP cannot get the responses from the CUD after a period of time specified in the bus protocol, protocol violation occurs. If the output signals from the CUD are misunderstood by a master IP, transaction invalidation occurs. In addition, some system operations may change the states of other system components when the CUD is interrupted, which can cause data inconsistency when the CUD is resumed. All the above problems have to be resolved in order to make correct interruption and resumption, which leads to the necessity of the PAM_IP_BUS.

We divide the bus (interconnect) protocols into three categories according to two conditions: whether there is an upper limit on the number of clock cycles for a transaction to complete (CD_1) and whether a transaction with a *prolonged* signals is supported (CD_2), where a prolonged signal is a signal from one side (master or slave) of a transaction to inform the other side that some requested data are not ready yet; the other side has to wait or retry in order to access the data. Thus a prolonged signal from a slave can instruct the master to re-access data from a specific point or to re-issue a request, and a prolonged signal from a master can instruct a slave to access or provide the required data later.

Table I summarizes three categories of the transactions under different (CD_1, CD_2) conditions and the interrupt handling schemes used to prevent transaction invalidation and protocol violation for each Category, where CD_1 = L (N) if there is an (no) upper limit on the number of clock cycles for a transaction and CD_2 = I (N) if there is a (no) prolonged signal for the bus protocol. Transactions in Category 3 are further divided into four groups, each with a different interrupt handling scheme. We next describe the operations of the PAM_CUD and the PAM_IP_BUS for each of these categories.

TABLE I. Classification of PAM_CUDs.

Category (CD1,CD2)	Condition (CD3,CD4)		(CD1, CD2_ CD3, CD4) Debug Scheme	Interrupt-ible
Category 1 (N,-)	N.A		Free_scheme	Yes
Category 2 (L,I)	N.A.		LI_scheme	Yes
Category 3 (L,N)	Transmitter/ Receiver (CD3)	Replicable transaction (CD4)	N.A.	N.A.
	Receiver	False	LN_RN	Yes
	Receiver	True	LN_RR	Yes
	Transmitter	True	LN_TR	Yes
	Transmitter	False	LN_TN	No

A. PAM_CUD

For transactions executed in Category 1 protocol ((CD_1, CD_2) = (N, -)), the corresponding scheme is called *Free_Scheme*. For this category when the CUD is interrupted within a transaction, the CUD_MA/SA will mask all handshake signals of the CUD. Thus, the transaction process is paused for the IP involved in the transaction, and the IP will wait for responses from the CUD. When the clock of the CUD is resumed, the interrupted transaction can be continued. Thus, the data between the CUD and IP are consistent.

Category 2 corresponds to (CD_1, CD_2) = (L, I), and the corresponding scheme is called *LI_Scheme*. Since the

interactive debug procedure may take seconds or even minutes for the user, the interrupted CUD will surely violate the limited-cycle constraint. In order not to violate the constraint, when the CUD is interrupted within a transaction in this category, the CUD_MA/SA will replace the CUD's role in the transaction by generating the prolonged signals according to the protocol specification. The IPs communicating with the interrupted CUD are notified that the handshake is interrupted and the current transaction needs to be prolonged.

The resumption process of transactions executed in Category-2 protocol depends on the role of the CUD in the broken transaction: master or slave. If the CUD is a master, since the CUD itself does not know that the transaction has been interrupted, the CUD_MA must issue the read/write transaction to the IP again. Once the IP accepts read/write request issued by the CUD_MA, the CUD_MA controls the resumption of the CUD, and then the CUD and the IP can complete the interrupted read/write transaction. If the CUD is a slave, its CUD_SA will issue the resumption signal to the master port before the CUD is resumed. Since the master port will keep on requesting the transaction when the CUD is interrupted, the CUD only needs to finish the remaining transaction once it is resumed.

Category 3 corresponds to (CD_1, CD_2) = (L, N). In order to maximize the efficiency of interruption handling, various PAM_CUD operations have been developed according to 1) the role of the CUD in the broken transaction (CD_3), and (2) whether the transaction can be *repeated* or not (CD_4), as described next.

In general, the completion of a transaction requires the correct cooperation among the master, the bus and the slave. If any of these three parts does not issue or respond to a transaction request, then the transaction cannot be completed. Thus a system can be "locked" by masking the handshake signals at the master or slave port, or by masking the grant signal of the bus. In this paper we shall call that ***a system is locked if no transaction can proceed due to the masking of the handshake or grant signals.***

We define a *replicable transaction* as one that satisfies the following two conditions.

1. The CUD is the master of the transaction, and
2. When the system is locked, the IP that is communicating with the CUD being interrupted will neither change the data that have been delivered or to be delivered by the transaction, nor make use of the data (such as providing the data to other IP or CUD). Hence when the CUD is resumed, the data can be resent to overwrite the previous data.

Transactions in Category 3 are classified into four groups according to the role of the CUD to be a transmitter (*T*) or receiver (*R*), and whether the transaction can be replicated (*R*) or not (*N*). The various schemes for PAM_CUD are implemented according to the classification as follows.

(1) *LN_**RN**_Scheme* (***R***eceiver/***N***on-replicable): The basic ideas to deal with this type of transaction are to use a buffer to store the data to be received by the CUD when interrupting the CUD and to "cheat" the IP involved in the transaction that the transaction will still be completed. The cheating is carried out by the CUD_MA/SA which will take over the role of the CUD in the transaction by sending required handshaking signal to the IP to continue the transaction. When the transaction is completed under the control of the CUD_MA/SA, the CUD_MA/SA will inform the PAM_IP_BUS to lock the system such that no further transaction can be executed. Then when the user finishes the debug process and wants to resume the system, the CUD_MA/SA will recover the clock of the CUD and deliver the data stored in the buffer to the CUD. Then it will inform the PAM_IP_BUS to unlock and resume the system.

(2) *LN_**RR**_Scheme* (***R***eceiver/***R***eplicable): In this scheme, the CUD_MA/SA will also take the role of the CUD to complete the undergoing transaction when the CUD is interrupted. However, unlike the previous scheme, in this scheme the data delivered in the transaction need not be stored. Instead it can be resent from the IP when the CUD is resumed since the transaction is replicable. When the current debug process is completed and the user wants to resume the system, the CUD_MA will provide the signal that can ask the IP to repeat the previous transaction such that the correct data to be delivered to the CUD by the previous transaction can be resent to the CUD. The control on the PAM_IP_BUS to recover the system is similar to the *LN_**RN**_Scheme*.

(3) *LN_**TN**_Scheme* (***T***ransmitter/***N***on-replicable): When the CUD is serving as a transmitter, it will deliver some data to the IP. If the CUD is interrupted during a transaction in which it serves as a transmitter, the receiver will never get the correct data in a valid transaction because the transaction is not replicable. Therefore, such transactions should not be interrupted.

(4) *LN_**TR**_Scheme* (***T***ransmitter/***R***eplicable): Since the CUD is a master, it can reissue a transaction request after it is resumed. Also since the receiver will not change or make use of the received data during the interrupted period, we can use CUD_MA to replace the CUD to provide some "dummy" data to the IP to complete the current transaction. The CUD_MA then can inform the PAM_IP_BUS to lock the system until the CUD is ready to resume. Before the resumption of the CUD, the CUD_MA will issue a write request that is the same as the interrupted transaction. Then when the CUD_MA is granted to use the bus, it will inform the PAM_IP_BUS to unlock the system and to accept data from the CUD.

B. PAM_IP_BUS

The main task of the *PAM_IP_BUS* is to lock the system operation by holding the current transaction or by preventing any initiation of new transactions during interruption.

When the PAM_IP_BUS receives signals from the PAM_CUD for the lock requirement, the PAM_IP_BUS will generate all the required signals to lock the system. This is done by masking the handshake signals at the master or slave port, or by masking the grant signal of the bus. Due to space

limitation the details of PAM_IP_BUS are not given in this paper.

V. Experimental Results

A. Industrial Case Study-Simulation

We use an industrial design of a JPEG decoder as the CUD to demonstrate the effectiveness and efficiency of the PAM. The normal operations of the JPEG core include setup, read/write and poll as described next. The ARM processor first sets up the core via its slave port. The master port of the JPEG core then issues read transactions to read a JPEG image from the memory. After processing the image, the JPEG core issues write transactions to write a YUV image to the memory. During the normal JPEG operations, the ARM processor keeps polling the slave port of the JPEG core to examine whether its normal operation is finished or not.

The simulation environment in which the PAM_IP_BUS are carried out at the master port of IP is shown in Fig. 2, which is based on the debug platform proposed in [10]. The *Test Bus* is used to transfer the debug control signals and debug data signals. The *Testbench* is used to control the breakpoint-based debug procedure via sending the cycle-based breakpoint into the TAM Controller.

To implement the PAM_CUD, the master port can request the slave port to extend data transmission time by signaling the prolonged signal (Category 2) defined in AMBA, HTRANS=BUSY. However, HTRANS=BUSY is optional in the AHB protocol; in case it is not implemented in the slave port, the PAM_CUD in the master port should belong to Category 3. In this experiment, the Category 3 PAM_CUD is implemented in the master port of JPEG, while Category 2 (LI_Scheme) PAM_CUD is implemented in the slave port.

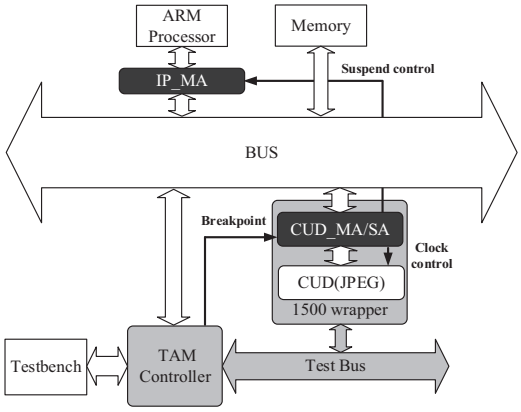

Fig. 2. Simulation environment with CUD_MA/SA and IP_MA from master ports of IP.

In Fig. 2, the lock of the master is achieved by the IP_MA of the ARM processor. During the interruption, the IP_MA can mask the bus request from the ARM processor. Extensive simulation on the system has been done and the results show that all the debug functions, including breaking the CUD, stopping the transactions of the whole system operation, dumping the internal states of the CUD, and resuming the system with a new breakpoint, can all be correctly carried out with the help of the protocol agents. Due to space limitation, the details of the experimental results are not given here.

B. Characteristics and Synthesis Results of CUDs

Five real circuits are also used as CUDs in our experiments, including a halftone, a JPEG, a DCT, a floating point unit and a bilinear pairing core [11]. The experiments use synthesized results of the five circuits by the TSMC 90nm CMOS technology. Two versions of the PAM are implemented whenever applicable: one supports replicable transactions while the other does not. The results show that the area overhead of the CUD ranges from 0.75% to 4.16% and the average is 2.2% if replicable transactions are not supported. With replicable transactions, the average area overhead is reduced to 1.1%, which is smaller since there is no need to provide data buffers for broken transactions. The performance impact of the PAM is 0% for most circuits, and only 2.02% at most. All clock cycles are interruptible under PAM for Category-1 and Category-2 protocol; for Category-3 protocol, the ratio of observable cycles can be as high as 44.52x of that of the transaction-level granularity approach.

V. Conclusions

The proposed PAM method facilitates breakpoint-based debug with cycle granularity. Experimental results show that the area overhead is very small and the performance penalty is negligible. The ratio of observable cycles is also much higher than that of the transaction-level granularity approach.

Acknowledgement:

This work was supported in part by the Ministry of Science and Technology of Taiwan under Contract MOST-102-2221-E-006-270-MY3.

References

[1] B. Vermeulen, "Functional Debug Techniques for Embedded Systems," *IEEE Design & Test of Comput.*, pp. 208-215, May-June 2008.

[2] F.-C. Yang, Y.-T. Lin, C.-F. Kao, I.-J. Huang, "An On-Chip AHB Tracer With Real-Time Compression and Dynamic Multiresolution Supports for SoC," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 4, April 2011.

[3] E. A. Daoud and N. Nicolici, "Real-Time Lossless Compression for Silicon Debug," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 9, Sep. 2009.

[4] E. A. Daoud and N. Nicolici,, "On Using Lossy Compression for Repeatable Experiments during Silicon Debug," *IEEE Trans. Comput.*, vol. 60, no. 7, July 2011.

[5] B. Vermeulen, T. Waayers, S. K. Goel, "Core-Based Scan Architecture for Silicon Debug," in *Proc. IEEE Int. Test Conf.*, 2002.

[6] B. Vermeulen, S. K. Goel, "Design for Debug: Catching Design Errors in Digital Chips," *IEEE Design & Test of Comput.*, pp. 35-43, 2002.

[7] K. Goossens, B. Vermeulen, R. van Steeden, M. Bennebroek," Transaction-Based Communication-Centric Debug," in *Proc. Int. Symp. Networks on Chip (NOCS)*, pp. 95-106, May 2007.

[8] B. Vermeulen and K. Goossens, "Interactive Debug of SoCs with Multiple Clocks," *IEEE Design & Test of Comput.*, vol. 60, no. 7, July 2011.

[9] H. Yi, S. Park, and S. Kundu, "On-Chip Support for NoC-Based SoC Debugging," *IEEE Trans. Circuits and Systems*, vol. 57, no. 7, July 2010.

[10] K.-J. Lee, S.-Y. Liang, A. Su, "A Low-Cost SOC Debug Platform Based on On-Chip Test Architectures," in *Proc. Int'l SOC Conf.*, pp. 161~164, 2009.

[11] OpenCores, Available: http://opencores.org/.