Easily Adaptable On-Chip Debug Architecture for Multicore Processors

Jing-Zhe Xu, Hyeongbae Park, Seungpyo Jung, and Ju Sung Park

Nowadays, the multicore processor is watched with interest by people all over the world. As the design technology of system on chip has developed, observing and controlling the processor core's internal state has not been easy. Therefore, multicore processor debugging is very difficult and time-consuming. Thus, we need a reliable and efficient debugger to find the bugs. In this paper, we propose an on-chip debug architecture for multicore processors that is easily adaptable and flexible. It is based on the JTAG standard and supports monitoring mode debugging, which is different from run-stop mode debugging. Compared with the debug architecture that supports the run-stop mode debugging, the proposed architecture is easily applied to a debugger and has the advantage of having a desirable gate count and execution cycle. To verify the on-chip debug architecture, it is applied to the debugger of the prototype multicore processor and is tested by interconnecting it with a software debugger based on GDB and configured for the target processor.

Keywords: JTAG, on-chip debugger, on-chip debug architecture, multicore processor debugging, monitoring mode debugging.

Manuscript received July 18, 2012; revised Sept. 17, 2012; accepted Oct. 1, 2012.

This work was supported by the Pioneer R&D Program for converging technology through the Korea Science and Engineering Foundation, funded by the Ministry of Education, Science and Technology, Rep. of Korea (M10711270001-08M1127-00110, Development of Displacement Sensing CMOS Circuit and Pattern Recognition System) and the Industrial Strategic Technology Development Program funded by the Ministry of Knowledge Economy, Rep. of Korea (No. 10039173, Development of System Semiconductor Technology for IT Fusion Revolution).

Jing-Zhe Xu (phone: +82 51 510 1702, kchuh@pusan.ac.kr), Seungpyo Jung (spyam@pusan.ac.kr), and Ju Sung Park (juspark@pusan.ac.kr) are with the Department of Electronics and Electrical Engineering, Pusan National University, Busan, Rep. of Korea.

Hyeongbae Park (baeya.park@gmail.com) is with the Department of R&D HW3 Team, Chips & Media Inc., Seoul, Rep. of Korea.

http://dx.doi.org/10.4218/etrij.13.0112.0487

I. Introduction

Today, the multicore processor is more attractive than the single-core processor and widely used in embedded systems. However, there are many problems that need to be solved, such as with interconnection, cache coherency, scheduling, synchronization, the programming model, application, and so on [1]-[3]. In addition to these problems, the debugging of a multicore processor is also a challenging task.

Some bugs of a system on chip (SoC) or its application programs appear only when executing the applications in real cases. So, having a debug unit to support debug functions is very necessary. Because of the dramatic increase in processor performance and the intrinsic lack of observability and controllability in multicore processors, an outside debug unit can no longer provide efficient debug capabilities, for example, in-circuit emulators or ROM monitors [4], [5]. In recent years, most processors employ an on-chip debug method that embeds a debug support logic into the target processor. The on-chip debug logic embedded in a processor works by interconnecting with a software debugger. It can allow the software developer to directly control the processor operation and examine the internal status, registers, and memories for debugging.

On-chip debug functionalities can be classified into three types, based on the supported debug method. The first type supports a run-stop method that stops the processor core on a desired breakpoint to inspect the core's state, for example, ARM's Embedded-ICE and MIPS's Extended JTAG [6]-[9]. The second type uses a real-time trace method that stores the debug data without halting the processor execution, for example, ARM's Embedded Trace Macrocell and MIPS's Program and Data Trace [10], [11]. The third type employs the monitoring method, which suspends the processor core to enter

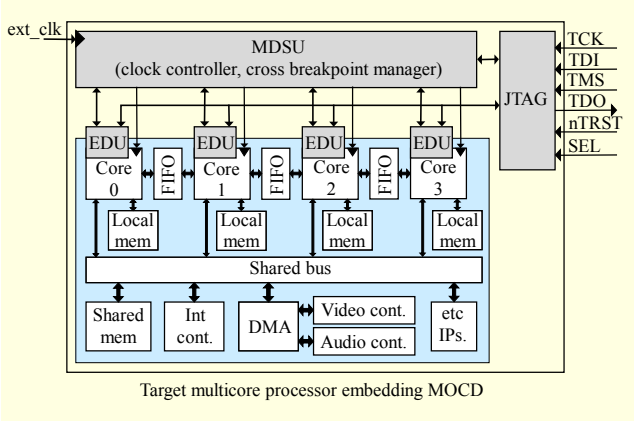

Fig. 1. MOCD architecture demonstrated by [14].

an exception to save the state information, for example, ARM's Monitor debug mode debugging [12]. The real-time trace method is a more powerful solution than the others, but it might cause huge hardware overhead because of the large memory for storing the debug information. Thus, the run-stop mode and monitoring mode debugging are more useful and flexible to implement.

Developing an on-chip-based processor debugger could be a difficult and complicated job that requires considerable time and effort. Therefore, easily adaptable and flexible on-chip debug architecture is needed for small companies and academic groups that develop their own customized processor cores for general or specific purposes. So, in this study, we propose an easily adaptable on-chip debug architecture for multicore processors. It employs the monitoring mode debugging method and is based on JTAG [13]. It supports debug functions, including breakpoint/watchpoint, single-step, register read/write, memory read/write, and debug/resume. These functions enable us to inspect the status of the multicore processor embedding the proposed debug architecture.

This paper is structured as follows. Section II reviews the existing debug architecture for multicore processors. Section III describes the proposed on-chip debug architecture for multicore processors. Section IV reports the implementation results, and conclusions are drawn in section V.

II. Related Work

In recent years, the smart devices are more popular for many people and most of them employ the multicore processors. Therefore, the task of debugging multicore processors plays an important role in the development of application systems. In several studies, researchers have proposed debug solutions for multicore processors. In the following paragraphs, we will review one of the solutions.

Figure 1 shows the on-chip debug architecture for the

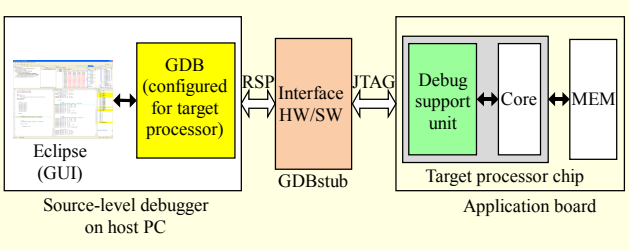

Fig. 2. Verification environment for on-chip debug architecture.

multicore processor proposed by [14]. Park and others [14] demonstrated that they focused on designing a flexible and scalable on-chip debug infrastructure that could provide the run-stop mode debugging method to multicore processors.

The multicore on-chip debug (MOCD) architecture consists of embedded debug unit (EDU) blocks, a multicore debug support unit (MDSU) block, and a JTAG block. Among the three, the EDU block is the most important block of the debug architecture because it can be embedded into each processor core and can be adapted to different multicore processors.

To adapt the MOCD to the other multicore processors, the processor cores must be modified to connect to the EDU block. If the cores are not the same, more work is required to adapt the MOCD to the multicore processor, even if the modification is minor. So, in this study, we propose an easily adaptable on-chip debug architecture for multicore processors and will describe it in section III.

To verify the proposed on-chip debug architecture, we implement a verification environment. Figure 2 gives an overview of the verification environment for the proposed debug architecture. It is composed of the debug support unit, the GDB, and the GDBstub. The debug support unit is the debug logic that employs the proposed on-chip debug architecture for multicore processors, the GDB is the source-level software debugger configured for the target processor, and the GDBstub is an interface hardware/software module [15]-[22].

Eclipse C/C++ development technology is used as the GUI tool for a software debugger. The GDBstub is the interface module for connecting the GDB to the debug support unit within the target through the GDB remote debug feature known as the remote serial protocol and JTAG.

III. Proposed On-Chip Debug Architecture

The proposed debug architecture, that is, the easily adaptable multicore on-chip debug (EA-MOCD), offers the processor control and the internal state inspecting functions to debug the multicore processor. The research is based on the MOCD, so we reuse some blocks designed by [14] for the EA-MOCD

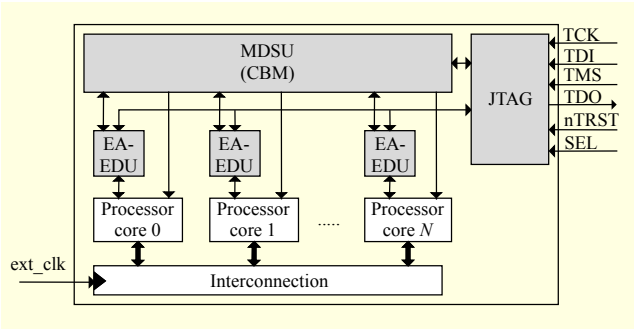

Fig. 3. Block diagram of EA-MOCD architecture with multicore processor.

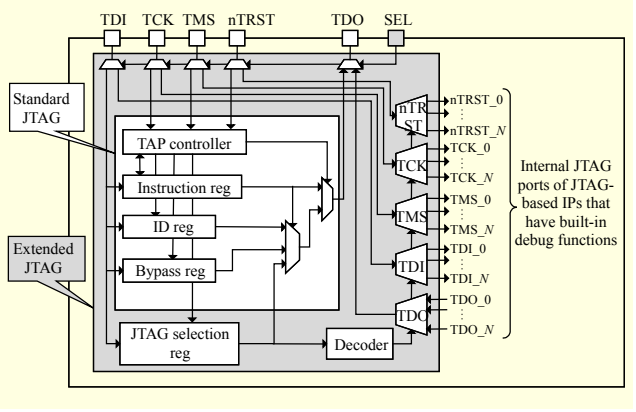

Fig. 4. Block diagram of extended JTAG block.

architecture, for example, the cross breakpoint manager (CBM) block and the comparator block. In this section, we will describe the structure and mechanism of the EA-MOCD. For the reused blocks, we will only do a short review.

The EA-MOCD consists of a JTAG, a multicore debug support unit (MDSU), and easily adaptable embedded debug unit (EA-EDU) blocks. The block diagram of the EA-MOCD is shown in Fig. 3 and is similar to that of the MOCD (see Fig. 1).

The JTAG block is used for debug interface and control. The original JTAG is not suitable for a multicore processor, so we modify the JTAG block to facilitate the debugging of the multicore processor through a single JTAG interface. As shown in Fig. 3, the EA-EDU blocks are connected with each processor core through the coprocessor interface to support debug functionalities. Multiple EA-EDU blocks can access each processor core separately. When one of the processor cores raises a breakpoint event, the other processor cores that are executing relevant tasks must enter the exception for debugging so that the possible debug point is not lost [23], [24]. For the concurrent debug operation, the EA-EDU blocks work in conjunction with the MDSU that includes the CBM module. A detailed view of each block is presented in the following paragraphs.

1. Extended JTAG

SoC includes so many IPs that there are not enough pins available for debugging. A JTAG (IEEE 1149.1 standard) uses only five I/O pins to interface the software debugger and hardware debugger. Therefore, we choose the JTAG protocol for transferring data between the hardware debugger and software debugger. The five JTAG ports support the debug interface between the software debugger and the EA-MOCD.

For application to the multicore processor, as [14] demonstrated, we extend the JTAG standard for our debug architecture. Figure 4 shows the detailed view of the extended JTAG block. It consists of one TAP controller, several registers for special purposes, a decoder, and additional functional logic. For the debugging of a multicore processor, we added several user defined JTAG instructions without any additional bit length of JTAG instruction register. While maintaining full IEEE 1149.1 compliance, the extended JTAG block uses only one TAP controller and JTAG connection to control and debug all JTAG-based IPs integrated in the multicore processor embedding the EA-MOCD.

The EA-MOCD supports the monitoring mode debugging instead of the MOCD's run-stop-type described by [14], and it does not use the inserting instruction method. So the scan chains of the EDU blocks are not necessary in our proposed debug architecture. Therefore, the gate count of the extended JTAG block in the EA-MOCD architecture is much less than it is in the MOCD.

2. MDSU

The MDSU, consisting only of the CBM module, is designed to allow embedded processor cores to be debugged concurrently, as reflected in the name "CBM." Even though the MDSU block only includes the CBM module, the reason why we use "MDSU" as the block name is that the study is based on the MOCD.

In a multicore processor system, the individual processor cores execute relevant multiple tasks in a parallel manner, while interacting with each other using task scheduling, synchronization, and communication via the interconnect method. So, while debugging one processor core, it is very important to control the rest of the processor cores to prevent commonly-encountered errors. Therefore, to effectively debug such multicore processor systems, cross breakpoint mechanisms are needed [25]-[28].

Figure 5 shows the relationship between the MDSU block and the neighboring blocks. The operation of the MDSU block

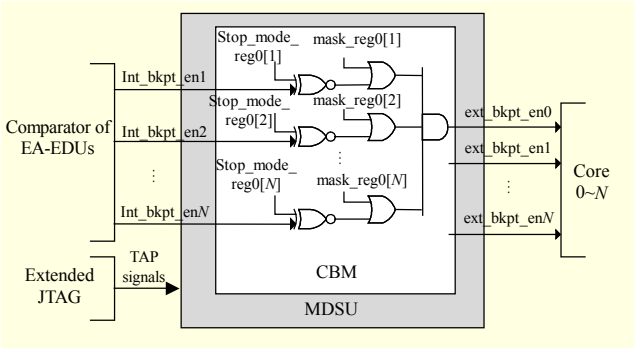

Fig. 5. Block diagram of MDSU.

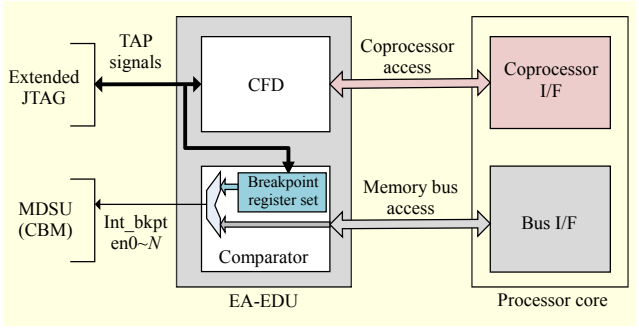

Fig. 6. Block diagram of EA-EDU.

is controlled by the extended JTAG block. The CBM module utilizes a number of the int_bkpt_en signals connected with the EA-EDU blocks and ext_bkpt_en signals connected with multiple processor cores, as shown in Fig. 5.

The int_bkpt_en signals generated by comparators of the EA-EDU blocks mean breakpoints have occurred. The ext_bkpt_en signals are used to force each processor core to enter the exception for debugging. The signals are determined by the combination of the int_bkpt_en signals and the internal configuration registers (stop mode value register and stop mode mask register), as shown in Fig. 5.

The MDSU does not include the clock controller module introduced by [14] because the EA-MOCD uses the monitoring mode debugging instead of the run-stop mode. Thus, even though there are two clock sources (system clock, JTAG tck clock), the EA-MOCD uses the individual clock domains that are different from the complex clock domains of the MOCD. Therefore, the proposed debug architecture is more reliable and robust than the MOCD in timing sensitivity. It is also easy to do the clock tree synthesis in the procedures of making silicon chips.

3. EA-EDU

The EA-EDU block is the most important block of the EA-MOCD because it plays a vital role in monitoring the processor core and transferring data (debug information and commands) between the core and software debugger through the JTAG protocol. Multiple EA-EDU blocks are connected to each processor core to support the debug capabilities in the multicore processor.

Figure 6 shows a block diagram of an EA-EDU block and its connection with the other blocks, such as the processor core, extended JTAG block, and the MDSU block. It consists of the comparator module and coprocessor for debug (CFD) module. The comparator module is designed to program and detect the breakpoint occurrence. The CFD module is a coprocessor used for transferring debug commands and information between the processor core and software debugger.

In part A of subsection III, we will briefly introduce the comparator module, studied by the researchers in [14]. In part B of subsection III.3, we will concretely describe the CFD, which is the key of the monitoring mode debugging method.

A. Comparator

As shown in Fig. 6, the comparator module includes the breakpoint register set and a comparator element. We can program one or two desired breakpoints in the breakpoint register set via the JTAG protocol for debugging. The comparator element is used for detecting the breakpoint occurrence by comparing the memory access signals of the processor core and the programmed breakpoint. When detecting a breakpoint occurrence, the comparator block activates the int_bkpt_en signal, which is connected with the MDSU block. In a multicore processor, multiple comparator modules generate some int_bkpt_en signals and the MDSU (CBM) block manages the input signals and generates the ext_bkpt_en signals (see Fig. 5).

B.CFD

The proposed debug architecture supports the debug functionalities using the monitoring mode debugging method. The processor core enters an exception for debugging when the ext_bkpt_en signal from the MDSU is enabled. So, the core has two operation modes, namely, the system mode and monitoring mode. The system mode is the normal operation mode of the core. The monitoring mode represents the exception for debugging. When an ext_bkpt_en signal is enabled, the core enters the monitoring mode, and we can execute the debug functions in this mode. The monitoring mode debugging method forces the processor cores to enter an exception for debugging the core and does not halt the target. Thus, the processor core is always operated by the system clock.

The CFD module is a coprocessor that can be connected

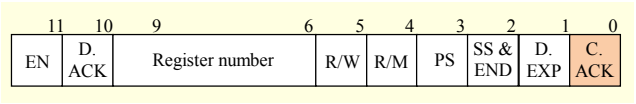

Fig. 7. MMCR bit pattern.

Table 1. MMCR bit functions.

Bits	Field	Function
[11]	EN	Specifies whether EA-MOCD is enabled (EN=1) or disabled (EN=0)
[10]	Debug acknowledge	Specifies whether processor core can access registers of CFD -D.ACK=1 for enabling access -D.ACK=0 for disabling access
[9:6]	Register number	Specifies number of desired registers for reading or writing
[5]	Read/write	Specifies whether it is read operation (R/W=0) or write operation (R/W=1)
[4]	Register/memory	Specifies whether it is operation for register (R/M=0) or memory (R/M=1)
[3]	Process status	Specifies whether it is operation for process status (PS=1) or not (PS=0)
[2]	Single-step & end	Specifies whether it is operation for single-step and debug end (SS&END=1) or not (SS&END=0)
[1]	Debug exception	Specifies whether processor core is in debug exception (D.EXP=1) or not (D.EXP=0)
[0]	Core acknowledge	Specifies whether software debugger can access registers of CFD -C.ACK=1 for enabling access -C.ACK=0 for disabling access

with each processor core to support the monitoring mode debugging on a multicore processor. It contains the coprocessor interface part and several registers for transferring debug information between the software debugger and the processor cores.

The CFD module is connected to the processor core via the coprocessor interface. Therefore, to adapt the EA-MOCD architecture to a multicore processor, it is necessary that the processor cores support the coprocessor interface. It is one of the restrictions for the EA-MOCD architecture. However, in our opinion, the addition of the coprocessor interface is considered to be acceptable for implementing a debug system of a multicore processor because it is an easy interface for processor design and most of the processors provide the coprocessor interface.

Figure 7 and Table 1 show the detailed information about the monitoring mode control register (MMCR). We can access the MMCR through the JTAG protocol and do the monitoring mode debugging by programming the MMCR according to the control mechanism as shown in Fig. 8. The processes of the debug functions are executed by transferring debug information through several registers (address register, read data register, and write data register of coprocessor) in the CFD.

According to the control mechanism, the EA-MOCD architecture supports the debug functions (breakpoint/watchpoint, single-step, register read/write, memory read/write, and debug/resume) via the CFD module. The operations of the debug functions are executed by the following four steps.

First, we program a desired breakpoint in the breakpoint register set in the comparator unit. That way, the comparator will activate the int_bkpt_en signal as 1 when it detects a breakpoint occurrence.

Second, the MDSU block receives the int_bkpt_en signal from the EA-EDU block and generates the ext_bkpt_en (see Fig. 5) signals for the processor cores. Then, it forces the processor cores to enter an exception for the monitoring mode debugging.

Third, during the monitoring mode, we can debug the target through the software debugger by executing the service routine of the exception for the monitoring mode. The service routine includes 49 core instructions and offers several operations (register read/write, memory read/write, process status read/write, single-step, and monitoring mode exit) according to the MMCR and control mechanism of the monitoring mode debugging.

Fourth, after finishing the debugging, the processor cores leave the exception and return to the system mode and the cores resume their previous states.

Due to the third step, it is necessary to involve the exception service routine (49 core instructions) in the instruction stream of the user for monitoring mode debugging. This is the other restriction for the EA-MOCD architecture. However, it is also an advantage for debugging because we can modify the exception service routine and the GDBstub software interface module (see Fig. 2) to generate new debug functions for the monitoring mode debugging without any hardware modification.

IV. Implementation Results

We apply our approach for the implementation of the debugger to a prototype multicore processor that contains four identical 32-bit RISC-type processor cores (core0 through core3). The processor cores are similar to the MIPS family of processors. They have a five-stage pipeline and use the Havard architecture.

Each processor core basically supports the coprocessor interface and memory interface as shown in Fig. 9. The signals for the coprocessor interface are prefixed with "COP_." The

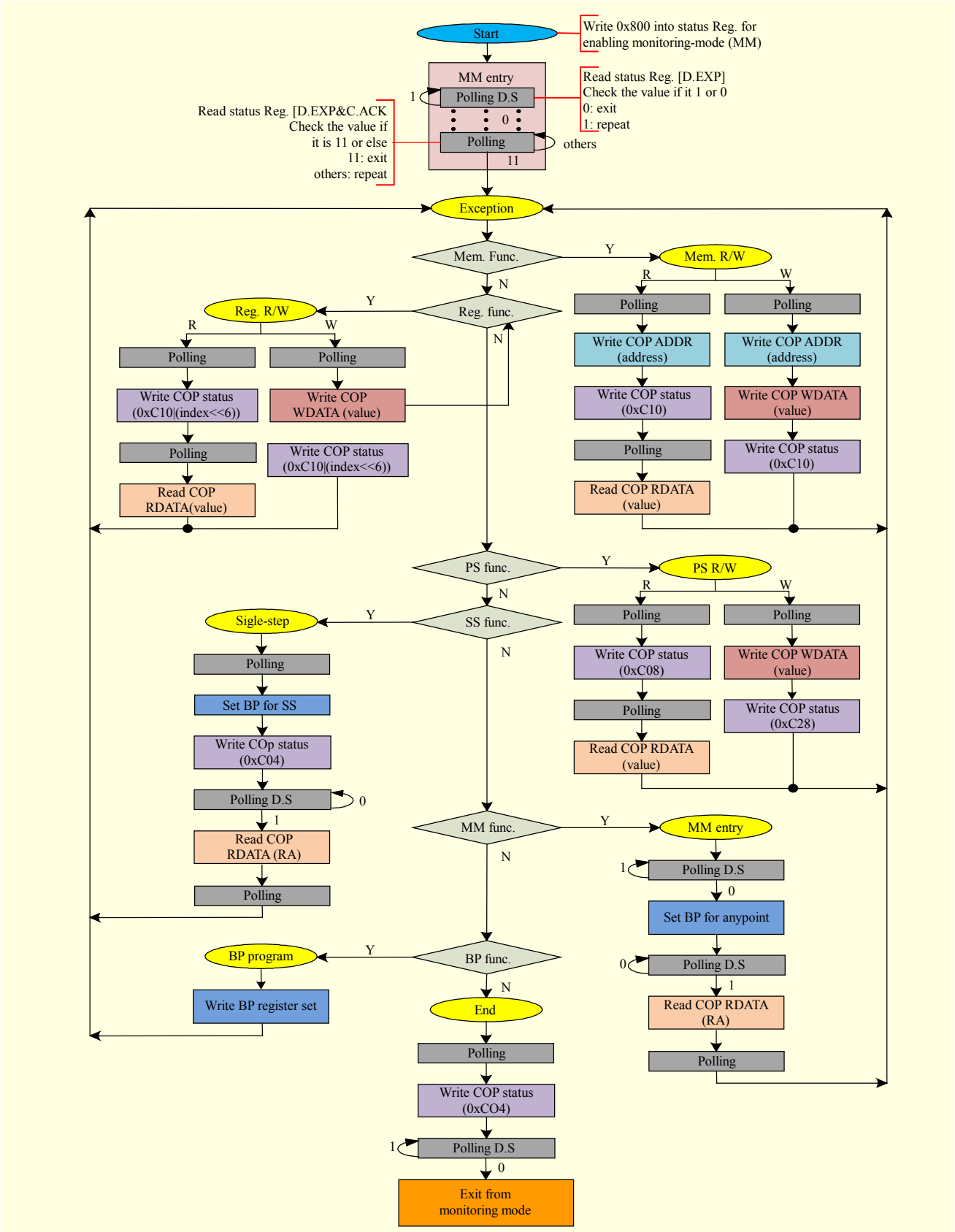

Fig. 8. Control mechanism of monitoring mode debugging.

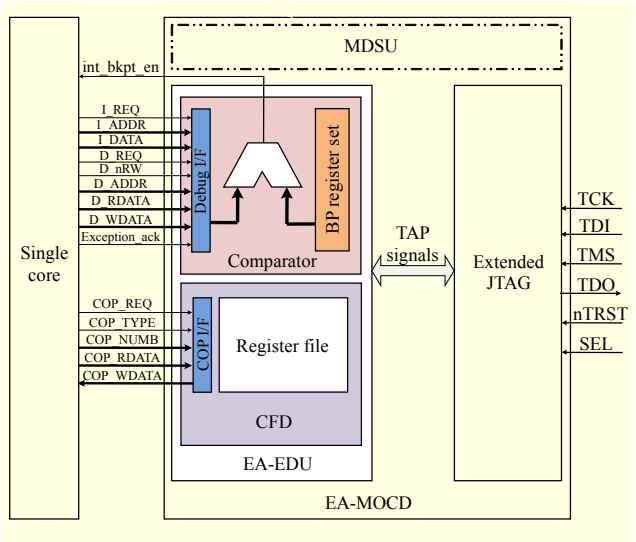

Fig. 9. Single core embedding EA-MOCD.

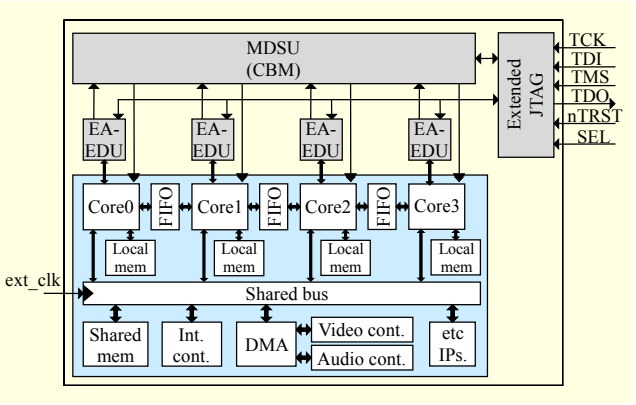

Fig. 10. Multicore processor embedding EA-MOCD.

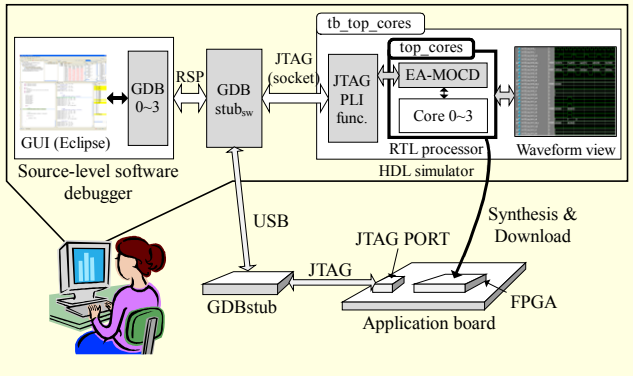

Fig. 11. Overall verification environment for EA-MOCD.

memory access signals use "I_" and "D_" as prefixes for instruction memory and data memory, respectively. The Exception_ack signal specifies whether the processor core enters a debug exception (Exception_ack=1) or not.

Figure 9 shows the detailed connections that allow the EA-MOCD to adapt to a single core. The EA-MOCD consists of a JTAG block, an MDSU block, and an EA-EDU block. But the MDSU (marked by dotted line in Fig. 9) is not necessary for adapting the EA-MOCD to a single core because there is only one core and we do not need to manage the breakpoints. So, unlike the int_bkpt_en0~*N* signals of the EA-MOCD for a multicore processor (see Fig. 5 and Fig. 6), the int_bkpt_en signal is connected directly to the processor core. Thus, we can easily adapt the EA-MOCD to a processor core if it supports the coprocessor interface and memory interface.

Figure 10 shows a detailed view of the multicore processor embedding the EA-MOCD. The EA-EDU blocks are connected with the MDSU block and extended JTAG block. The JTAG block controls the MDSU block and EA-EDU blocks to process the debug functions by a single TAP. The EA-EDU blocks monitor the state of instruction and data buses and detect the desired breakpoint occurrence. The MDSU block receives the int_bkpt_en0~3 signals from each EA-EDU block and generates the ext_bkpt_en0~3 signals for core 0 to core 3. Furthermore, the whole EA-MOCD block is connected with the multicore processor by the coprocessor interface and memory interface.

Figure 11 shows the overall verification environment for the EA-MOCD as introduced in section II. We conduct a two-stage verification procedures: functional level simulation by a simulator and FPGA prototyping level verification. Functional level simulation is time-consuming but enables us to apply various desired methods and algorithms in the verification process. FPGA verification tests the applications in real-time, so it can verify the timing sensitivity and exception cases.

As shown in Fig. 11, the RTL multicore processor model (top_cores) could be connected with four GDB source-level software debuggers that are configured for the target processor cores via the GDBstub software interface module and JTAG programming language interface (PLI) functions of the test-bench module (tb_top_cores). Thus, the method makes it possible to debug the multicore processor embedding the EA-MOCD block running on the simulator at the source level through the GDB along with its powerful software debug functions, instead of the complicated simulation waveform view. The solution provides a more efficient environment for simulation. Due to the functional level of simulation, we prove not only the EA-MOCD's debug functionalities but its reliability. However, functional level simulation does not consider the gate delay, exceptions, and real time, so we conduct the FPGA prototyping level verification later.

We execute the synthesis and P&R procedures for the design (top_cores) and implement the design in the FPGA on the application board connected with a software debugger through

Table 2. Comparison result of gate count.

Functional blocks		Area (# of 2-input NAND gates)			
		For 1 core		For 4 cores	
		MOCD	EA-MOCD	MOCD	EA-MOCD
JTAG	TAP	2,323	2,323	2,323	2,323
	Scan chain	6,355	None	25,420	None
MDSU	Clock controller	14	None	64	None
	CBM	0	0	553	553
EDU / EA-EDU	Comparator	13,127	13,127	52,508	52,508
	SMC	268	None	1,072	None
	CFD	None	863	None	3,452
Total		22,087	16,313 (↓ 21.6%)	81,940	58,836 (↓ 28.2%)

Table 3. Comparison result of execution cycles for debug functions.

Debug functions		Execution cycles (# of TCK cycles)	
		MOCD	EA-MOCD
16 register values	Read	2,122	2,168 (↑ 2.2%)
	Write	2,563	2,411 (↓ 5.9%)
16 memory values	Read	3,839	3,912 (↑ 1.9%)
	Write	4,618	4,133(↓ 10.4%)
Single-step		13,606	7,598 (↓ 44.2%)

the GDBstub hardware/software interface modules. To verify the EA-MOCD block, we test all the debug functions at the desired breakpoint, such as breakpoint programming and detection, debug/resume, single-step, register read/write, memory read/write, and variable read/write.

Table 2 shows the comparison result of the gate count between the proposed EA-MOCD and the MOCD introduced by [14]. We perform the synthesis procedure using a commercial 90-nm CMOS cell library. The gate count of the EA-MOCD is less than that of the MOCD at about 21.6% and 28.2% for a single-core processor and a four-core processor, respectively.

The comparison result of the execution cycles for several debug functions is shown in Table 3. For example, the single-step debug function is executed by the following sequence: programming the breakpoint for single-step, restoring the previous register values, exiting to system mode from debug mode, re-entering debug mode, and saving current register values. For each executing cycle of the breakpoint programming, debug mode entering and exiting is similar between the EA-MOCD and the MOCD. In the procedures of restoring and saving the register values, the MOCD uses the inserting instruction method synchronized by 32 cycles of the JTAG tck clock and the EA-MOCD employs the exception method synchronized by one cycle system clock. The period of the JTAG tck clock is much longer than that of the system clock. So, for the single-step debug function, the EA-MOCD architecture requires less execution cycles than the MOCD does, at about 44.2%. Table 3 indicates that the EA-MOCD method is better than the MOCD regarding debug speed.

The EA-MOCD architecture has the following two restrictions. First, the processor cores must provide a coprocessor interface for connecting with the EA-MOCD. Second, the user must embed 49 instructions for the debug service routine into the user's instruction stream. In this aspect, the EA-MOCD has disadvantages. However, it is still superior to the MOCD regarding gate count and execution cycles.

To adapt the MOCD to a new multicore processor, we must modify the processor cores slightly, as demonstrated by [14]. It is also necessary to modify the GDBstub software interface module because the MOCD uses the inserting instruction scheme for debugging while other processor cores use a different instruction set. Therefore, regarding adapting the debug architecture to different multicore processors, the EA-MOCD is definitely superior to the MOCD because the EA-MOCD does not require any modification whereas the MOCD requires some modification of the processor cores and software debugger.

The proposed EA-MOCD architecture is considered significantly more powerful than the MOCD in terms of supporting the same debug functionalities.

V. Conclusion

This paper presented our proposed on-chip debug architecture for multicore processors. It supports monitoring mode debugging and such debug functions as breakpoint/watchpoint, single-step, register read/write, memory read/write, and debug/resume.

The EA-MOCD architecture consists of a JTAG block, an MDSU block, and multiple EA-EDU blocks. The JTAG block is extended for multicore processor debugging and it controls the rest of the debug units (MDSU and EA-EDU) to execute the debug functions. The MDSU block works in conjunction with the multiple EA-EDU blocks and JTAG block. It supports synchronous and concurrent debugging among the processor cores. The EA-EDU blocks are the debug support unit that can be connected with each processor core by a coprocessor interface and bus interface supported by the core. There is not a modification job for connecting the EA-EDU blocks to

processor cores. So, the EA-MOCD architecture can be easily adapted to different multicore processors that support the coprocessor interface without any change of the processor cores and the software debugger.

To verify the proposed debug architecture, we applied the EA-MOCD architecture to a prototype multicore processor system and implemented an overall verification environment including a GDB-based software debugger, a GDBstub, and an EA-MOCD block. We verified the reliability of the EA-MOCD at an RTL simulation level and FPGA prototyping level.

The proposed EA-MOCD architecture has two restrictions, which were discussed in section IV. However, the EA-MOCD architecture has the advantage of having a desirable gate count and execution cycle for processing the debug functions. It also can offer developers significant help in adopting it as a debug solution for multicore processors.

References

[1] W. Wolf, A. Jerraya, and G. Martin, "Multiprocessor System-on-Chip (MPSoC) Technology," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 10, Oct. 2008, pp. 1701-1713.

[2] T. Dorta et al., "Overview of FPGA-Based Multiprocessor Systems," *Int. Conf. Reconfigurable Comput. FPGAs*, 2009, pp. 273-278.

[3] G. Martin, "Overview of the MPSoC Design Challenge," *43rd ACM/IEEE Design Autom. Conf.*, 2006, pp. 274-279.

[4] Y. Zorian, E.J. Marinissen, and S. Dey, "Testing Embedded-Core-Based System Chips," *Computer*, vol. 32, no. 6, June 1999, pp. 52-60.

[5] A.B.T. Hopkins and K.D. McDonald-Maier, "Debug Support Strategy for Systems-on-Chips with Multiple Processor Cores," *IEEE Trans. Comput.*, vol. 55, no. 2, Feb. 2006, pp. 174-184.

[6] ARM Ltd. Embedded-ICE Block Specification. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0180a/DDI0180.pdf

[7] MIPS Technologies Inc. EJTAG Trace Control Block Specification. Available: http://www.mips.com/securedownload/ index.dot?product_name=/auth/MD00148%2D2B%2DETCB%2DSPC%2D01.04.pdf

[8] JTAGPPC Controller. Available: http://www.xilinx.com/products/intellectual-property/jtagppc_cntlr.htm

[9] L. Lian et al., "Design and Implementation of A Debugging System for OpenRISC Processor," *2nd ASID Conf.*, Aug. 2008, pp. 368-371.

[10] ARM Ltd. ETM (Embedded Trace Marcocell) Block Specification. Available: http://www.arm.com

[11] PDtraceTM Interface Specification, MD00136, May 14, 2003. Available: http://www.mips.com

[12] ARM Ltd. Monitor Debug-Mode Block Specification. http://www.arm.com

[13] IEEE Std. 1149.1a-1993, "Test Access Port and Boundary-Scan Architecture," Piscataway, NJ: IEEE, 1993.

[14] H. Park et al, "On-Chip Debug Architecture for Multicore Processor," *ETRI J.*, vol. 34, no. 1, Feb. 2012, pp. 44-54.

[15] R. Stallman, R. Pesch, and S. Shebs, "GDB User Manual: Debugging With GDB (The GNU Source-Level Debugger)," Free Software Foundation.

[16] B. Gatliff, "Embedding with GNU: The GDB Remote Serial Protocol," Red Hat Developer Network (RHDN), 1999.

[17] M. Tan, A Minimal GDB Stub for Embedded Remote Debugging, 2002. Available: http://www1.cs.columbia.edu/~sedwards/classes/2002/w4995-02/tan-final.pdf

[18] S. Shebs, GDB: An Open Source Debugger for Embedded Development, Red Hat, 2000.

[19] Robert Pizzi, "GNU GDB Internal Architecture," 1993.

[20] J. Gilmore and S. Shebs, GDB Internals, Cygnus Solutions, 2004. Available: www.gnuarm.com/pdf/gdbint.pdf

[21] J. Bennett "Howto: Porting the GNU Debugger: Practical Experience with the OpenRISC 1000 Architecture," Nov. 2008. Available: http://www.embecosm.com/download/ean3.html

[22] Open On-Chip Debugger. Available: http://openocd.berlios.de/web/

[23] CoreSight On-Chip Trace and Debug Specification. Available: http://www. arm.com

[24] B. Vermeulen and S. Bakker, "Debug Architecture for the En-II System Chip," *IET Comput. Digit. Techn.*, vol. 1, no. 6, Nov. 2007, pp. 678-684.

[25] N. Stollon et al., "Multi-core Embedded Debug for Structured ASIC Systems," *Proc. Design Con*, 2004.

[26] R. Leatherman and N. Stollon, "An Embedded Debugging Architecture for SoCs," *IEEE Potentials*, vol. 24, no. 1, 2005, pp. 12-16.

[27] S. Tang and Q. Xu, "A Debug Probe for Concurrently Debugging Multiple Embedded Cores and Inter-core Transactions in NoC Based Systems," *Proc. Asia South Pacific Design Autom. Conf.*, Seoul, Rep. of Korea, 2008, pp. 416-421.

[28] L. Fiorin, G. Palermo, and C. Silvano., "MPSoCs Run-Time Monitoring through Networks-on-Chip," *Proc. Conf. Design, Automation Test Europe*, 2009, pp. 558-561.

Jing-Zhe Xu received his BS in electronic communication engineering from Yanbian University of Science and Technology, Yanji, Jilin, China, and his MS in electronics engineering from Pusan National University, Busan, Rep. of Korea, in 2005 and 2008, respectively. He is currently working toward his PhD in electronics engineering at Pusan National University. His research interests include microprocessor design, multicore platform implementation, and on-chip debug architecture.

Hyeongbae Park received his BS in telecommunication engineering from Dongseo University, Busan, Rep. of Korea, in 2004 and his MS and PhD in electrical engineering from Pusan National University, Busan, Rep. of Korea, in 2006 and 2012, respectively. He is currently in the R&D HW3 team at Chip & Media Inc., Seoul, Rep. of Korea. His research interests include application-specific processor design, multicore architecture processor design for multimedia application, and on-chip debug architecture.

Seungpyo Jung received his BS and MS in electronics engineering from Pusan National University, Busan, Rep. of Korea, in 2007 and 2009, respectively. He is currently working toward his PhD in electronics engineering at Pusan National University. His research interests include microprocessor design and multimedia platform implementation.

Ju Sung Park received his BS in electronics engineering from Pusan National University, Busan, Rep. of Korea, in 1976, his MS in electrical engineering from KAIST, Seoul, Rep. of Korea, in 1978, and his PhD in electrical engineering from the University of Florida, Gainsville, FL, USA, in 1989. From 1978 to 1991, he was with ETRI, Daejeon, Rep. of Korea, where he worked as a principal research engineer and as the manager and director of the IC Design Group. While at ETRI, he designed several bipolar analog ICs and was in charge of developing VCR ICs, CMOS 8-bit microprocessors, and telecommunication chips. In 1991, he joined the Electronics Department, Pusan National University, where he is now a professor of electronics engineering. His current research interests are microprocessor and DSP core design, platform design and application, and multimedia algorithm implementation by hardware and software co-design.