On-Chip Debug Architecture for Multicore Processor

Hyeongbae Park, Jing-Zhe Xu, Kil Hyun Kim, and Ju Sung Park

Because of the intrinsic lack of internal-system observability and controllability in highly integrated multicore processors, very restricted access is allowed for the debugging of erroneous chip behavior. Therefore, the building of an efficient debug function is an important consideration in the design of multicore processors. In this paper, we propose a flexible on-chip debug architecture that embeds a special logic supporting the debug functionality in the multicore processor. It is designed to support run-stop-type debug functions that can halt and control the execution of the multicore processor at breakpoint events and inspect the possible causes of any errors. The debug architecture consists of the following three functional components: the core debug support block, the multicore debug support block, and the debug interface and control block. By embedding this debug infrastructure, the embedded processor cores within the multicore processor can be debugged simultaneously as well as independently. The debug control is performed by employing a JTAG-based scanning operation. We apply this on-chip debug architecture to build a debugger for a prototype multicore processor and demonstrate the validity and scalability of our approach.

Keywords: Design-for-debug, on-chip debug, processor debugger, multicore processor debugging, JTAG-based debugging.

Manuscript received Mar. 21, 2011; revised July 3, 2011; accepted July 18, 2011.

Hyeongbae Park (phone: +82 51 510 1702, hbpark@pusan.ac.kr), Jing-Zhe Xu (kchuh@pusan.ac.kr), Kil Hyun Kim (k85123@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.

http://dx.doi.org/10.4218/etrij.12.0111.0172

I. Introduction

In recent years, the multicore processor containing two or more processor cores on a single chip has been widely used in embedded systems that require high processing power, such as networking, communication, signal processing, and multimedia systems. The design of a multicore processor involves many difficult problems that need to be solved, such as interconnection, cache coherency, scheduling, synchronization, and programming model. [1]-[3]. Apart from these complexities, the multicore architecture of the processor presents new challenges in debugging.

Because of the dramatic increase in processor performance and the intrinsic lack of internal-system observability and controllability in highly integrated multicore processors, conventional debugging approaches that implement debug functions from outside the processor chip can no longer provide efficient debugging capabilities, for example, in-circuit emulators or ROM monitors [4]. To address these difficulties, most multicore processors employ an on-chip debug method, also known as design-for-debug (DfD), with a special hardware debug function embedded in the processor chip. As multicore processors can have diverse structures and architectures with a wide range of processor cores, the implementation of effective debug functions tailored to the target multicore processor is certainly one of the most important design challenges. There are several types of on-chip debug approaches that focus on different processor operations and support different debug functionalities.

On-chip debug functionalities can be classified into two types based on the supported debugging method. The first type features a run-stop (intrusive) scheme that uses execution control to start a processor and then stops it on a breakpoint placed at a point of interest to allow inspection of the processor's state, for example, ARM's Embedded-ICE and

MIPS's Extended JTAG (EJTAG) [5]-[8]. The second type uses a real-time trace (non-intrusive) scheme that stores the debug information (such as program flow or memory access address/value) into the internal or the external memory without halting the processor execution, for example, ARM's Embedded Trace Macrocell (ETM) and MIPS's Program and Data Trace (PDtrace) [9], [10]. This method complements the run-stop debugging by providing additional information about the timing behavior of the processor operation [11].

Although, the real-time trace scheme is a more efficient debugging solution than the run-stop scheme, with respect to supported debugging capability, it may not be the appropriate debug solution in some cases due to complexity and hardware overheads. It requires on-chip or off-chip memory to store the traced debug data and a set of trace ports to transfer debug data out of the chip at high speeds, and it also needs an efficient compression/decompression hardware to reduce the amount of debug data [6], [8], [12].

Different from the above conventional processor-centric debug solutions focusing on the processor's computational operations and its interaction with main memory, in recent years, new research has introduced communication-centric debug solutions for network-on-chip (NoC)-based multicore processors to make the interactions between the intellectual property (IP) blocks via the communication architecture observable and controllable [13]-[15].

Among the diverse debug approaches, the processor-centric debug solution employing the run-stop scheme gives the developer more power to observe the functional behavior of the processor core at the exact possible erroneous point, in a controlled manner. This debug method allows each embedded processor core of the multicore processor to be controlled and accessed independently. In addition, it must be able to provide special debug capabilities to handle many of the debugging issues that are specific to a multicore architecture processor, such as interoperability, communication, and synchronization between the embedded processor cores.

In this study, we propose on-chip debug architecture for the multicore processor that supports processor-centric debug operations in a run-stop fashion. It is capable of controlling the target multicore processor to perform debug operations. Further, it also capable of observing the multicore processor's internal status to inspect the root cause of erroneous behavior. Our main objective is to develop a flexible and practical processor-centric run-stop debug solution that can be readily integrated into multicore processors at the register-transfer level (RTL) with minimum modifications, rather than defining an extensive standard for a wide range of debug applications.

This paper is organized as follows. In section II, we review the existing debug approaches for multicore processors. In section III, we present a brief overview of the proposed on-chip debug architecture for multicore processors. The three functional blocks of the debug architecture are described in detail in sections IV, V, and VI. The implementation results are explained in section VII, and the concluding remarks are made in section VIII.

II. Related Work

Most of the recent multicore processors employ an on-chip debug method that adds special debug-support IP blocks into the design. Researchers have proposed several on-chip debug methods for a multicore processor. First Silicon Ltd. presented an integrated debug platform including a software debugger called multicore embedded debug that contained an on-chip instrument block as the core debug supporting module, HyperDebug block for debugging intercore communication, and HyperJTAG [16], [17]. To address a diverse range of debug requirements, it introduced system-level debug solutions that can monitor and trace embedded cores during normal operation. ARM proposed debug methodologies called CoreSight for ARM-based and AMBA-based multicore processor [5], [6], [18], [19]. As the CoreSignt is designed for an ARM-based system-on-chip (SoC), it cannot provide an appropriate debug solution when the target multicore processor does not employ ARM cores and an AMBA system bus or when embedded processor cores do not have built-in debug functions.

In recent years, debug standardization research has been widely carried out in an effort to bring together IP core providers, semiconductor manufacturers, and vendors of debug tools in five organizations: the Nexus 5001, Mobile Industry Processor Interface (MIPI) Test and Debug, IEEE P1149.7, IEEE P1687, and Open Core Protocol International Partnership (OCP-IP) Debug working groups [20]. As the groups' joint goal is to standardize the debug interfaces between different IP cores on a chip as well as between different chips, and the external debug equipment and tools, they may not offer practical debug solutions for the multicore architecture processor.

There have been several studies on on-chip debug implementations based on the following standards for multicore processors: Nexus 5001, IEEE std. 1500, and OCP-IP [21]-[24]. To make the embedded processor cores compatible with these standards, a designer must develop an interface block, a so-called wrapper, to wrap around the embedded processor cores to connect them using the common interface protocol. This wrapper circuit must include a core debug supporting logic to allow the embedded cores to be controlled and debugged according to the required debug

operations. This standard-based debug implementation scheme can be useful in developing the debug solutions of certain multicore processors, which employ a complex intercore communication method such as NoC, because it can facilitate the connection of debug units between the embedded cores. However, it is possible that this approach may be more difficult and complex when the multicore processor does not adopt the standard-based interface method.

As mentioned above, depending on the implementation method and the supported debug functionality, there are various kinds of debug solutions for multicore processors. In this study, we focused on designing a flexible and scalable on-chip debug infrastructure that can provide the processor-centric run-stop debug functionalities on multicore architecture processors. We also discuss its application method.

III. Multicore On-Chip Debug Architecture: Overview

In this section, we present an overview of the proposed processor-centric debug architecture, called multicore on-chip debug (MOCD). The details follow in the later sections. Figure 1 shows the block diagram of the debug architecture and the relationship between the functional units on the multicore processor with respect to the important signals.

The debug architecture consists of three functional units: the core debug support block, the multicore debug support block, and the debug interface and control block. The core debug support block, called embedded debug unit (EDU), is embedded into each processor core within the target multicore processor to support run-stop debug functionalities. By embedding this EDU block, each processor core can be accessed and debugged independently by using basic debug functions, common to most processors and IEEE 1149.1 JTAG-based debug implementations, including breakpoints and watchpoints, single stepping, read/write register, and read/write memory. Multiple EDU blocks can monitor each embedded processor core's operation separately. When one of the processor cores raises a breakpoint event, other processors that are executing relevant tasks must be stopped immediately following the condition that raised the event so that the possible debugging point is not lost [25], [26]. For this concurrent debug operation, the EDU blocks work in conjunction with the multicore debug support block, called the multicore debug support unit (MDSU) that includes the clock controller module and cross breakpoint manager module. The MDSU block can control the execution of all the embedded processor cores and other hardware IP according to several configuration registers. As shown in Fig. 1, JTAG is used for the debug interface and control [27]. This JTAG block has extended features to facilitate debugging of a multicore processor, which allows all

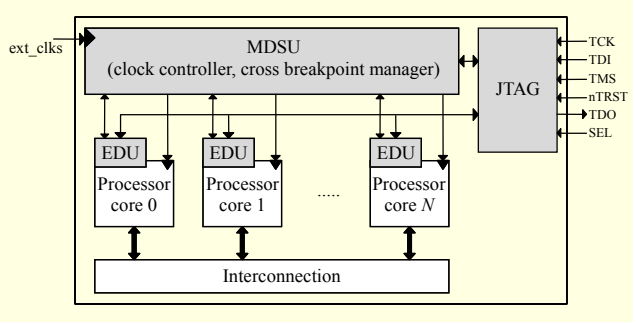

Fig. 1. Overall organization of MOCD architecture on a multicore processor that is composed of three functional blocks: EDU, MDSU, and JTAG.

embedded processor cores to be accessed only through a single JTAG connection.

Since the proposed debug architecture supports run-stop-type debugging operations, the processor execution is stopped at a point of interest to inspect its internal status. Therefore, each processor core has two modes of operation, namely, the run mode and the stop mode. The run mode represents the processor's normal operation. Once a breakpoint condition occurs, the processor enters into the stop mode, indicating that the processor core is halted for debugging. When the debug operations are complete, the operation mode is switched back to the run mode. For this run-stop operation, as shown in Fig. 1, the internal clock input of each processor core is supplied from the clock controller of the MDSU block and not directly connected to the external clock (ext_clk). This approach enables the internal clock of each processor core to be controlled for performing the debug operations. Due to this run-stop debug operation, the processor cores need to be modified and extended to manage this mode-switching operation at the breakpoint address.

In the following sections, we will describe the function and relationship of the three functional units (EDU, MDSU, and JTAG) and provide an overview of how to integrate these with a given multicore processor.

IV. Embedded Debug Unit

The EDU block is a core-debug-supporting hardware block that can be embedded into each processor core to support debug capabilities on a multicore processor. Figure 2 shows the structure of the EDU block and the connection signals that are required to interface it with the processor core.

The EDU block can be divided into the following three functional modules: comparator, switch mode controller (SMC), and IEEE 1149.1 boundary scan chain. The comparator module is designed to detect the breakpoint condition and the scan chain module is used to inspect the

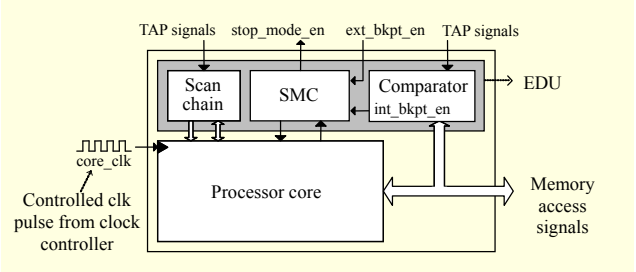

Fig. 2. EDU block is a core debug supporting hardware block that can be embedded into the processor core.

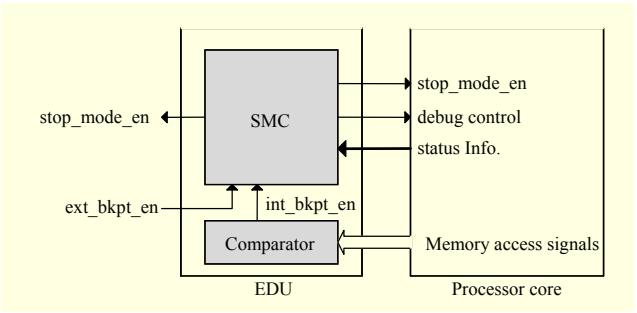

Fig. 3. Required interface signals for mode-switching operation: stop_mode_en, debug control, and status Info. signals.

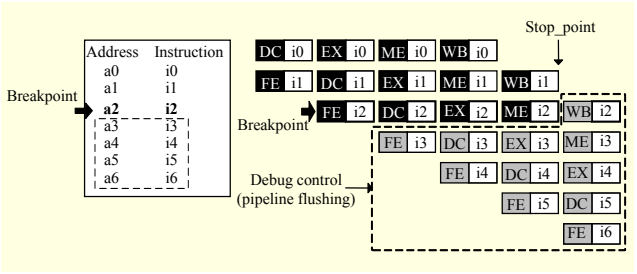

Fig. 4. Mode-switching operation of processor core at breakpoint address (a2).

processor core's internal status. The SMC module serves to control and interact with the processor core through several interface signals, to handle the mode-switching operation properly (between the stop mode and the run mode).

Because of this mode-switching operation, the processor core needs to have appropriate functions and interface signals that enable the EDU block to control the core's execution for performing debug operations. In the following three subsections (IV.1, IV.2, and IV.3), we describe the functions of the three modules. In subsection IV.4, we discuss how to interface them with a processor core.

1. Comparator

The comparator module involves a number of registers and one comparator element. It can monitor and detect the breakpoint condition, by comparing the memory access signals of the processor core (including the address buses, the data buses, and the memory control signals) against the programmed breakpoint register values that can be accessed via the JTAG protocol [28]. The breakpoint registers include the address registers, data value registers, control registers, debug status registers, and mask registers. The address bus and the data bus can be masked according to the address mask registers and the data mask registers, respectively, to exclude a specific address area or a specific data value [5].

When a breakpoint condition occurs, the internal breakpoint enable signal (int_bkpt_en) is enabled. These signals are connected with the SMC module to indicate whether the breakpoint condition is satisfied or not (see Fig. 2). The int_bkpt_en signal can be disabled by the control register.

2. Switch Mode Controller

In run-stop-style debugging, when the breakpoint event occurs, the processor core enters into the special debug mode, that is, the stop mode. This indicates that the processor core must be halted immediately following the condition that raised the event, so that the possible debugging point is not lost. Therefore, the EDU block must be capable of managing this mode-switch operation of the processor core. As shown in Fig. 3, the SMC module in the EDU block functions to handle this mode-switching operation between the run mode and the stop mode through several debug purpose interface signals (status Info., debug control, and stop_mode_en). The SMC keeps track of the breakpoint event through the int_bkpt_en signal from the comparator module, and asserts the stop_mode_en signal when the processor is in the appropriate stop condition. The ext_bkpt_en input signal is used to force the processor core into the stop mode so as to support concurrent debug operations between embedded processor cores (this will be discussed in section V.2).

For the program execution not to be affected in normal running mode (the run mode), the processor core has to be in an appropriate operational status before entering into the stop mode [29]. Figure 4 illustrates an example of the mode-switching operation at a breakpoint address (a2). The mode-switching operation must be activated only after the previous address instructions (i1) of the breakpoint address have been completed and the subsequent instructions running on each pipeline stage (i3, i4, i5, and i6) have been canceled.

The SMC can detect the completion time of the previous instruction (stop_point) through a set of status Info. signals [28]. This completion time is different depending on the kind of instruction, for example, multiple load/store or repeat instructions. Moreover, for a conditional-branch-type instruction, the mode-switching operation must be carried out

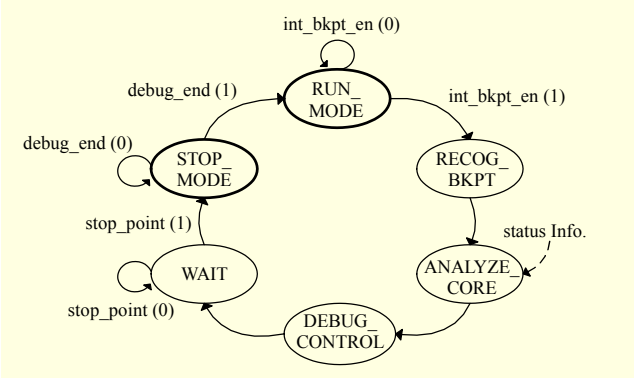

Fig. 5. State diagram of SMC module for handling mode-switch operation.

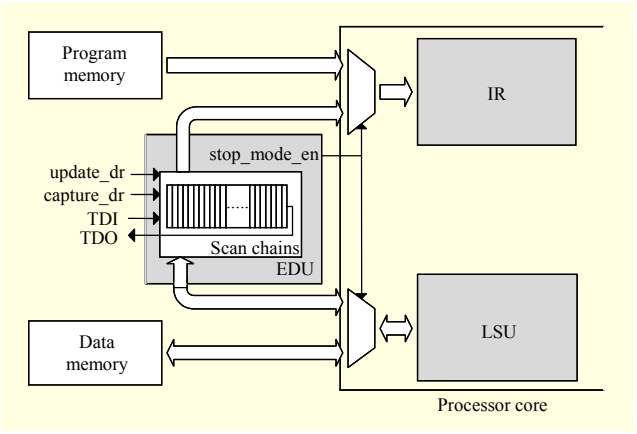

Fig. 6. Utilization of boundary scan chain to inspect internal status of processor core. It is used for inserting instructions into IR and reading or writing register value from/to LSU.

only if the condition is false. Because processor cores usually have some special performance-improving features that can affect the mode-switching operation, the types of status Info. signals can be different depending on the architectural features of the target processor core. In addition, the SMC can cancel the pipeline operations of the subsequent instructions by using the debug control signal. To enable this SMC to handle these debug operations, the processor core must support the debug purpose interface signals and functions.

The operation of the SMC module can be illustrated by state diagrams, as shown in Fig. 5. The SMC detects the breakpoint event by the int_bkpt_en signal in the RUN_MODE state, and recognizes the breakpoint condition (address breakpoint or data value breakpoint) in the RECOG_BKPT state. The exact mode-switching time can be detected through the status Info. signals in the ANALYZE_CORE state, and the debug control is carried out in the DEBUG_CONTROL state. When the stop_point signal is enabled in the WAIT state, the processor core enters into the stop mode while asserting the stop_mode_en signals in the STOP_MODE state. When debugging is completed (debug_end is enabled), the processor core returns to the run mode (RUN_MODE state).

The stop_mode_en and ext_bkpt_en signals can be interfaced with debug purpose signals of external hardware IPs, such as interrupt request signals or debug request signals, to support application-specific debug operations that can facilitate the debugging of the target multicore processor.

3. Scan Chain

As described in previous subsections, the breakpoint operation is performed by the comparator module and the SMC module. The single-step debugging operation can be executed by setting the breakpoint at the next address. The register read/write and memory read/write debugging operations are performed by using the IEEE 1149.1 boundary scan chains of the JTAG block. Figure 6 shows the utilization of the scan chain to inspect the registers and memory. The scan chain resides in parallel with the memory access bus of the processor core. When the target processor core is in the stop mode (stop_mode_en is enabled), the scan chain is connected to the instruction register (IR) and to the load and store unit (LSU) (The IR block is a register that is used to store an instruction temporarily in the FETCH pipeline stage, and the LSU block is a functional element that serves to manage all memory read (load) and memory write (store) operations).

This feature allows a debug purpose instruction and data value to be shifted into the scan chain in the Shift-DR state, and loaded into the IR and the LSU in the Update-DR state, through the test data input (TDI) port. While the processor core is stopped, this inserted instruction and data value can be executed by the controlled clock pulse of the clock controller (it will be discussed in section V.1). Also, the execution results can be captured into the scan chain from LSU in the Capture-DR state, and shifted out in the Shift-DR state through the test data output (TDO) port. This scheme resembles the operating method of ARM's Embedded-ICE [5], [29]. For example, the debugging process involves the following three steps for reading the R0 register value from the target processor core via the scan chain.

Step 1. Insert the store register (STR) instruction into the scan chain. After entering into the stop mode, the STR instruction is inserted into the IR though the TDI port.

Step 2. Execute the STR instruction until the MEMORY pipeline stage. The inserted STR instruction is then executed by the controlled clock pulses from the clock controller. The R0 register value is loaded into the LSU block at the MEMORY pipeline stage.

Step 3. Read the R0 register value through the scan chain.

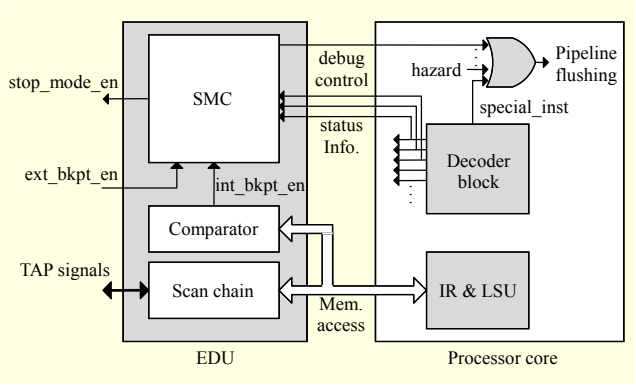

Fig. 7. Extensions of processor core to interact with EDU block to support built-in debug operations.

The R0 value in the LSU block is then shifted out through the TDO port.

4. Extension of Processor Core

Figure 7 shows the extension of the processor core to support the debug purpose interface signals and functions to work with the EDU block. To detect the mode-switching time, the processor core can provide several status Info. signals that are constituents of the control signals from the decoder block; these are the decoding results of the previous instruction of the breakpoint address (refer to section IV.2). In general, the processor core has some functional blocks for bubbling the pipeline, also known as a pipeline break, pipeline stall, or pipeline flush, to prevent data, structural, and branch hazards from occurring by inserting NOPs into the pipeline before the next instruction (which would cause the hazard) is executed [30]. The debug control signal that is used to cancel the subsequent instructions of the breakpoint address is added to the pipeline flush block. In addition, as shown in Fig. 7, the boundary scan chains are added to the core's program/data memory access bus. Alternatively, if the processor core has its own scan chain in the memory access bus, this can be used for the EDU's debugging operation.

These types of modification do not affect the given processor core. As discussed above, because the debug purpose interface signals and functions (status Info. and debug control) can be implemented by minor extensions of the decoder block and the pipeline flush block, the interfacing of the EDU block does not require the design of additional functional blocks in the processor core. Since general processor design typically includes both types of functional blocks (decoder blocks and pipeline flush blocks), this form of extension is possible for most processor core models at RTL. Therefore, the EDU block can be interfaced with different processor cores at RTL with minor modifications that do not adversely affect the original processor core design.

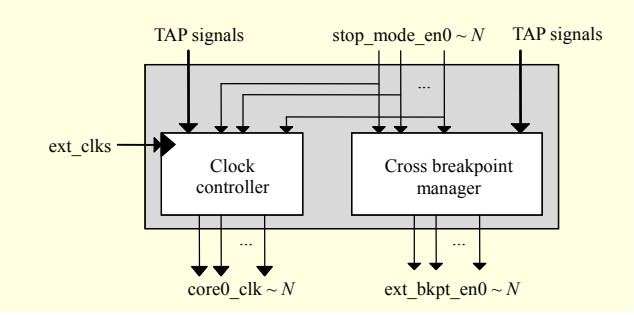

Fig. 8. Block diagram of MDSU that consists of clock controller module and cross breakpoint manager module.

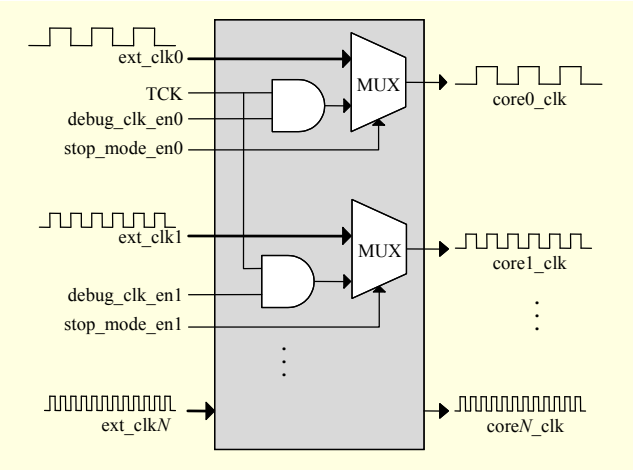

Fig. 9. Block diagram of clock controller; core0_clk, core1_clk, and core*N*_clk represent internal clock inputs of *N* embedded processor cores, respectively.

V. Multicore Debug Support Unit

In this section, we describe the MDSU that works in conjunction with the multiple EDU blocks of the embedded processor cores (Fig. 1). The MDSU, consisting of the clock controller module and the cross breakpoint manager module, is designed to allow embedded processor cores to be debugged concurrently through the stop_mode_en signals from each EDU block, as shown in Fig. 8.

1. Clock Controller

Basically, for the run-stop-type debugging, the on-chip debug infrastructure should support two functions, that is, stopping the processor core's execution at the breakpoint address, and returning back to normal running mode. This run control of the processor core has been traditionally implemented by controlling its internal clock [5], [31]-[33]. Figure 9 shows the implementation of the clock controller

hardware for our multicore debug infrastructure. To perform the debugging operations, the clock controller can control the execution of each embedded processor core by gating the external clock input.

As shown in Fig. 9, the clock controller allows the internal clock of each processor core to be disabled independently during the processor chip's normal operation. If the core0 processor hits a breakpoint, the stop_mode_en0 signal is enabled. This forces the internal clock input (core0_clk) to be disabled by gating the external clock input (ext_clk0) through a 2×1 multiplexer, while the core0 processor enters the stop mode. When the debug operation is completed, the stop_mode_en0 signal again becomes disabled; the ext_clk0 signal passes directly to the core0_clk, and the core0 processor returns back to the normal running state (the run mode).

Additionally, in the stop mode, the clock controller can generate a number of controlled clock pulses while the internal clock input is stopped by stop_mode_en signals. As shown in Fig. 9, the controlled clock pulses for each processor core can be issued by using the test access port (TAP) controller signals, test clock (TCK), and several debug_clk_en signals (debug_clk_en0, debug_clk_en1, etc.). One of debug_clk_en signals is enabled when the TAP controller is in the Run Test/Idle state, and a scan chain selection register, which is a user-defined JTAG register, is configured to select the processor core being debugged (this will be discussed in section VI). The generated clock pulses are used to execute the instructions inserted from the boundary scan chain to inspect the internal status of the processor core in the stop mode, as discussed in section IV.3.

2. Cross Breakpoint Manager

In a multicore architecture processor, the individual embedded 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. Therefore, if one processor core hits a breakpoint, the other processor cores executing relevant tasks may have possible errors resulting in undesired behaviors. Therefore, all the embedded processor cores may need to be debugged concurrently. To effectively debug such multicore-specific operations, so-called cross breakpoints or cross triggering mechanisms are needed [16], [17].

In our debug architecture, a cross breakpoint manager (CBM) module is designed to support this concurrent debugging operation. Figure 10 shows the CBM module's operational mechanism by using I/O signals. The CBM module utilizes a number of the stop_mode_en signals (stop_mode_en0 to stop_mode_en*N*) and ext_bkpt_en signals

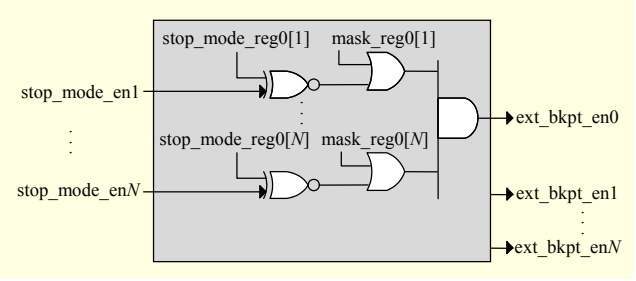

Fig. 10. Block diagram of CBM that employs cross breakpoint mechanism between embedded processor cores.

(ext_bkpt_en0 to ext_bkpt_en*N*), which are connected to multiple EDU blocks of the embedded processor cores (see Fig. 1).

As described in section IV.2, the ext_bkpt_en0 is used to force the core0 processor into the stop mode. This ext_bkpt_en signals are determined by the combination of the stop_mode_en signals from other EDU blocks (stop_mode_en1 to stop_mode_en*N*) and the internal configuration registers (stop mode value register, stop mode mask register), as shown in Fig. 10. In addition, the CBM module can be extended to support target-specific debug requirements by connecting it to different hardware IP, for example, debug purpose signals or interrupt request signals [16], [17].

VI. Extended JTAG Structure

The entire on-chip debug infrastructure is controlled and programmed through an IEEE 1149.1 JTAG. Although it was originally developed for I/O testing, the IEEE 1149.1 JTAG has become a default interface method for other on-chip test/debug features, including embedded debug blocks commonly available for processor cores [12], [34]. To provide specific debug/test functions depending on the target application, the JTAG can be modified and extended by supporting additional user-defined JTAG instructions or adding special purpose registers and functional blocks [5], [7], [35], [36].

We extended the JTAG function for our debug infrastructure. Figure 11 shows the schematic overview of the extended JTAG structure. It includes a number of boundary scan chains for individual EDU blocks (residing in parallel with the memory access bus), a TAP controller, several special purpose registers, and additional hardware logic (four multiplexers, a de-multiplexer, and a decoder). Further, to support the debugging of a multicore processor, the TAP controller has been extended with several user-defined JTAG instructions. This JTAG logic has two extended features to facilitate debugging of a multicore processor.

First, as shown in Fig. 11, the scan chains of the EDU blocks can be accessed separately because they are not concatenated into a single long serial chain; thus, the debugging speed can be

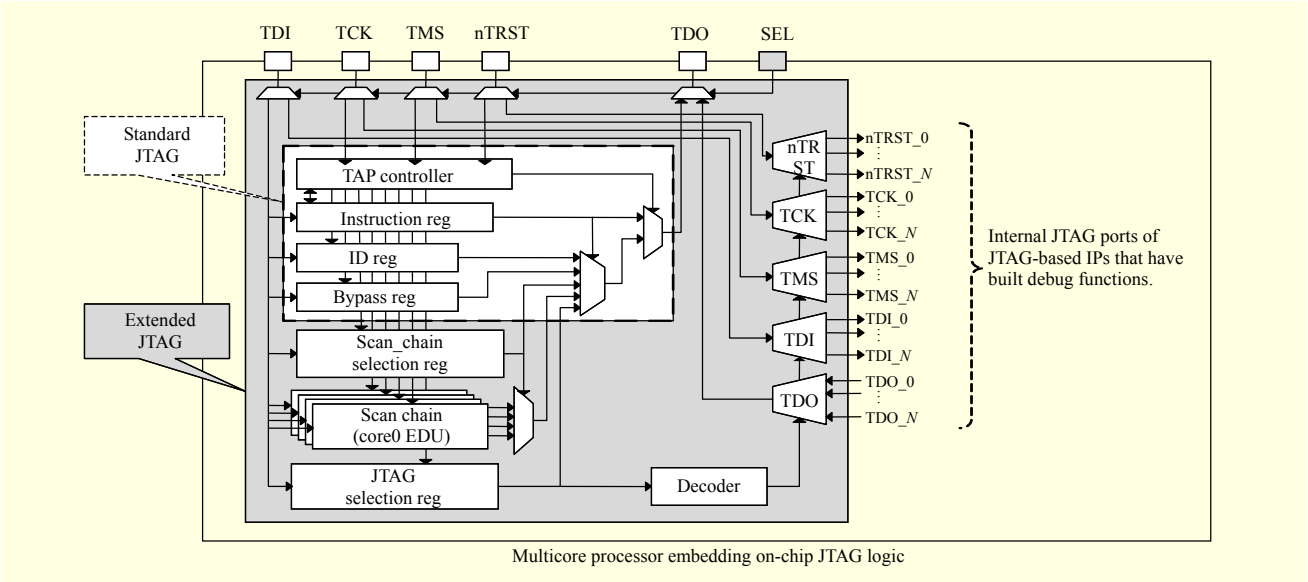

Fig. 11. Block diagram of on-chip JTAG block that has extended features for multicore processor debugging; scan chains within JTAG are interfaced with individual core's memory access bus (refer to section IV.3).

increased. These scan chains have their own numbers to facilitate easy access to them. To select particular scan chains to route to the TAP, the JTAG contains an additional register, called the scan chain selection register, and supports a private JTAG instruction, sel_scan_chain, with all required public JTAG instructions [5], [10]. When the TAP controller is in the Update-IR state, the sel_scan_chain instruction is loaded into the JTAG instruction register. In the Shift-DR state, the scan chain number is shifted into the scan chain selection register. This extended feature allows the proposed MOCD architecture to be accessed over a single JTAG connection.

Secondly, much hardware IPs has built-in debug capabilities based on JTAG, such as ARM, MIPS, and PowerPC. If the hardware IPs are integrated with the target multicore processor embedding MOCD infrastructure, additional JTAG pins are required to allow each of them to be debugged [35], [36]. However, we extended JTAG to allow multiple JTAG-based IPs on a single chip to be accessed via a single JTAG connection. The proposed JTAG structure includes additional hardware (JTAG selection register, decoder, multiplexer, and demultiplexers), and a JTAG port, SEL, and, it supports the additional JTAG instruction, sel_jtag. Multiple internal JTAG ports of hardware IPs can be connected with a single external JTAG port by this extended JTAG functionality. First, the JTAG selection register is configured to select the JTAG-based IP which is to be accessed. The decoder determines several outputs of the multiplexer (TDI, TMS, TCK, nTRST) and an input of the demultiplexer (TDO) according to the JTAG selection register. The SEL port remains "low" while MOCD infrastructure is accessed. However, the SEL port must be "high" when accessing the JTAG-based hardware IPs, after configuring the JTAG selection register. Once the SEL port is "high," a set of internal JTAG ports (selected by the decoder block according to the JTAG selection register) are directly connected to the external JTAG ports. This approach is simple yet powerful because it eliminates the need for an additional JTAG pin through only a small extension of the TAP controller while maintaining full IEEE 1149.1 compliance.

VII. Implementation Results

To verify the proposed MOCD architecture, we applied it to a prototype multicore processor that is designed for multimedia streaming applications. The target multicore processor contained four identical 32-bit RISC-type processor cores (core0 to core3) that had some architectural features similar to MIPS family processors. The processor had three FIFOs for communication between the processor cores and several peripheral IPs.

Figure 12 illustrates the implemented multicore processor incorporating the MOCD infrastructure. To embed the EDU block, we modified the processor cores such that they have the debug purpose functions and interface signals that serve to control the mode-switching operation between the run and the stop modes, and we inserted boundary scan chains to the memory access signals. The EDU blocks are connected with the exterior MDSU block and JTAG block. For debug control, the external clock input (ext_clk) of the multicore processor is connected with the clock controller module of the MDSU, and the internal clock input of each embedded processor is supplied

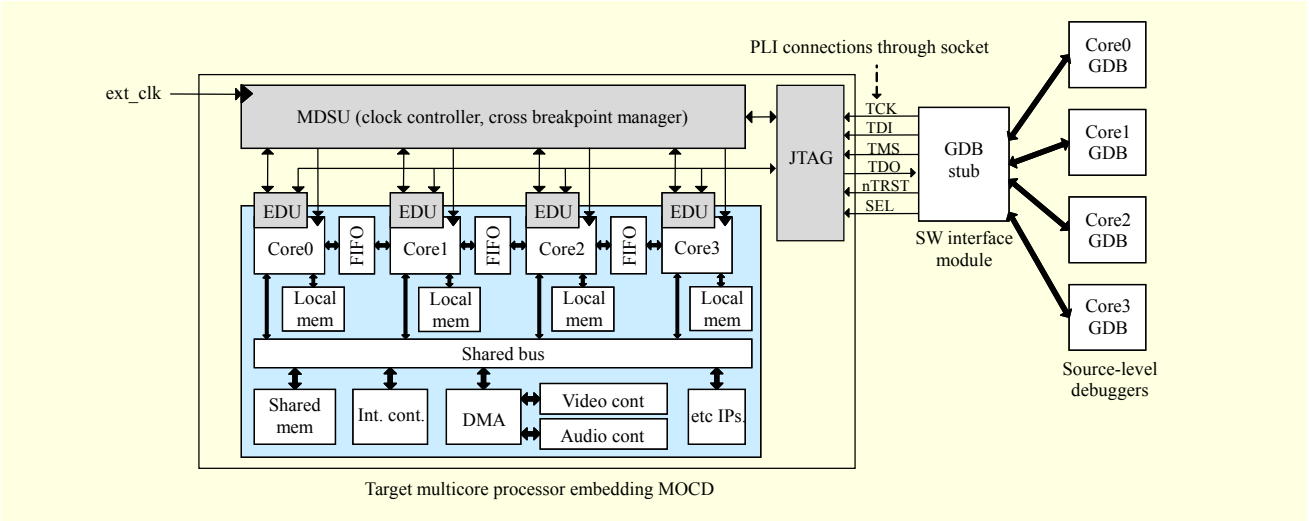

Fig. 12. Verification environments on RTL simulation level using multicore processor embedding MOCD, four GDB source-level debuggers, and GDBstub.

by this clock controller. This work did not significantly affect the design of the multicore processor. Furthermore, as discussed in section IV.4, the embedding of the EDU block did not require large extensions of the processor cores, which could have an adverse effect on their original design.

As shown in Fig. 12, the verification was performed on RTL simulation level. We interfaced the multicore processor running on the HDL simulator with four GDB source-level debuggers that are configured for the target processors via the GDBstub software interface module. We implemented GDBstub to support the remote debugging capability of GDB. It communicates with GDB using remote serial protocol (RSP) and generates JTAG format signals to control the on-chip JTAG block through a TCP/IP socket connection [37], [38]. To enable the GDBstub module to approach the JTAG block running on HDL simulator, we developed a set of user-defined program language interface (PLI) functions that allow JTAG TAPs to be approached via a TCP/IP socket channel. The RTL multicore processor model is linked to the external four GDBs via the GDBstub interface module by employing these user-defined PLI functions.

Table 1 shows the gate counts of the proposed MOCD infrastructure in a commercial 90-nm CMOS library. The gate count overhead is about 22,087. However, this gate count increases with the number of processor cores because of the embedding of more EDU blocks and scan chains in the target multicore processor. In our prototype multicore processor that contains four EDU blocks and four scan chains, the total gate count overhead is about 81,940 gates. This hardware overhead is considered to be acceptable in comparison with existing multicore debug solutions that support similar debug functionalities [11].

Table 1. Area overhead of MOCD infrastructure in commercial 90-nm CMOS library.

Functional blocks		Area (# of 2-input NANDs)	
		1 core	4 cores
EDU	Comparator	13,127	52,508
	SMC	268	1,072
MDSU	Clock controller	14	64
	CBM	0	553
JTAG	TAP controller	2,323	2,323
	Scan chain	6,355	25,420
Total		22,087	81,940

VIII. Conclusion

As multicore architecture processors become more complex and sophisticated, the importance of debugging will continue to increase because of the intrinsic lack of the internal chip's observability and controllability. In this study, we proposed a flexible MOCD architecture that is designed to support the processor-centric run-stop-type debugging functions, including run, halt, single step, and breakpoint, together with concurrent debug operations between embedded processor cores.

The MOCD architecture consists of three functional blocks, that is, EDU, MDSU, and JTAG. We designed a core-debug-supporting logic, EDU, that can be embedded into each processor core of the multicore processor and make it possible to debug them separately. These multiple EDU blocks are

connected with the MDSU block to provide the synchronous and concurrent debug functions. All of the debug control is performed by a single TAP connection of the extended JTAG block.

The MOCD architecture can be applied to different multicore processors at RTL. Depending on the target multicore processor, each block can be extended to fit it. We defined the required extensions and modifications of the multicore processor for the run-stop debug operations, such as the connection method of clock signals and debug request/acknowledge signals, and described how to embed the three functional blocks into it. We applied the MOCD infrastructure to a simple prototype multicore processor, and we successfully verified its validation and scalability at RTL-simulation level. The MOCD architecture can offer developers a significant amount of flexibility in finding the optimum processor-centric debug solutions 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] A.B.T. Hopkins and K. 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, doi:10.1109/TC.2006.22.

[5] ARM Ltd. Embedded-ICE Block Specification. Available: http://www.arm.com

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

[7] JTAGPPC Controller. Available: http://www.xilinx.com

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

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

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

[11] B. Vermeulen, "Functional Debug Techniques for Embedded Systems," *IEEE Design Test Comput.*, vol. 25, no. 3, 2008, pp. 208-215.

[12] H.F. Ko, A.B. Kinsman, and N. Nicolici, "Design-for-Debug Architecture for Distributed Embedded Logic Analysis," *Very Large Scale Integr. Syst.*, vol. 13, 2010, pp. 1-14.

[13] K. Goossens et al., "Transaction-Based Communication-Centric Debug," *Proc. Int. Symp. Netw. On-Chip*, May 2007.

[14] B. Vermeulen, K. Goossens, and S. Umrani, "Debugging Distributed-Shared-Memory Communication at Multiple Granularities in Networks on Chip," *Proc. Int. Symp. Netw. On-Chip*, 2008, pp. 3-12.

[15] B. Vermeulen and K. Goossens, "A Network-on-Chip Monitoring Infrastructure for Communication-Centric Debug of Embedded Multiprocessor SoCs," *Int. Symp. VLSI Design, Automation Test*, 28-30 Apr. 2009, pp. 183-186.

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

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

[18] ARM11MPCore Specification. Available: http://www.arm.com

[19] W. Orme, "Debug and Trace for Multicore SoCs," Sept. 2008. Available: http://www.arm.com

[20] B. Vermeulen et al., "Overview of Debug Standardization Activities," *IEEE Design Test Comput.*, vol. 25, no. 3, May 2008, pp. 258-267.

[21] H. Yi, S. Park, and S. Kundu, "On-Chip Support for NoC-based SoC Debugging," *IEEE Trans. Circuits and Syst. Part I: Regular Papers*, vol. 57, no. 7, 2010, pp. 1608-1617.

[22] S. Tang and Q. Xu, "A Multi-core Debug Platform for NoC-Based Systems," *Proc. Design, Autom. Test Europe Conf. Exhibition*, Apr. 2007, pp. 1-6.

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

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

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

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

[27] IEEE Std. 1149.1a-1993, "Test Access Port and Boundary-Scan Architecture," *IEEE*, 1993.

[28] H. Park et al., "Design of On-Chip Debugging System Using GNU Debugger," *IEEK*, vol. 46, no. 1, 2009, pp. 24-38.

[29] I.-J. Huang et al., "A Retargetable Embedded In-circuit Emulation Module for Microprocessors," *IEEE Design Test Comput.*, vol. 19, no. 4, July-Aug. 2002, pp. 28-38.

[30] D.A. Patterson and J.L. Hennessy, *Comput. Organization Design*, 4th ed., Morgan Kaufmann Publishers, 2009, p. 336.

[31] H. Hao and R. Avra, "Structured Design-for-Debug-the

SuperSPARCTMII Methodology and Implementation," *Proc. Int. Test Conf.*, 1995, pp. 175-183.
[32] H. Hao and K. Bhabuthmal, "Clock Controller Design in SuperSPARCTMII Microprocessor," *Proc. Int. Test Conf. Comput. Design,* 1995, pp. 124-129.
[33] G.J. van Rootselaar and B. Vermeulen, "Silicon Debug: Scan Chains Alone Are Not Enough," *Int. Test Conf.*, 1999, p. 892.
[34] G.R. Alves and J.M.M. Ferreira, "From Design-for-Test to Design-for-Debug-and-Test: Analysis of Requirements and Limitations for 1149.1," *Proc. 17th IEEE VLSI Test Symp.*, Los Alamitos, CA: IEEE CS Press, 1999, pp. 473-480.
[35] A. Hopkins and K. McDonald-Maier, ''Debug Support for Complex Systems On-Chip: A Review,'' *IEE Proc. Comput. Digit. Tech.*, vol. 153, no. 4, July 2006, pp. 197-207.
[36] B. Vermeulen, T. Waayers, and S. Bakker, "IEEE 1149.1-Compliant Access Architecture for Multiple Core Debug on Digital System Chips," *Proc. Int. Test Conf.*, Oct. 2002, pp. 55-63.
[37] R. Stallman, R. Pesch, and S. Shebs, *Debugging with GDB: The GNU Source-Level Debugger*, Free Software Foundation.
[38] Open On-Chip Debugger. Available: http://openocd.berlios.de/web/

Hyeongbae Park received the BS in telecommunication engineering from Dongseo University and the MS in electrical engineering from Pusan National University, Busan, Rep. of Korea, in 2004 and 2006, respectively. He is currently working toward the PhD in electronics engineering at Pusan National University. His research interests include application-specific processor design, multicore architecture processor design for multimedia application, and on-chip debug architecture.

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

Kil Hyun Kim received the BS from the Department of Control and Instrumentation Engineering from Pukyong University, Busan, Rep. of Korea, in 2010. His research interests include design of high-performance processor and on-chip debug architecture.

Ju Sung Park received the BS in electronics engineering from Pusan National University, Busan, Rep. of Korea, in 1976, the MS in electrical engineering from KAIST, Seoul, Korea, in 1978, and the PhD in electrical engineering from University of Florida, Gainsville, 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.