2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
DECEMBER 6 - 7, 2022

Generic Core-Monitor for Hardware/Software Co-Debugging targeting Emulation Platform

Shreya Morgansgate (Shreya.Morgansgate@infineon.com)

Dr. techn. Johannes Grinschgl (Johannes.Grinschgl@infineon.com)

Dr. Ing. Djones Lettnin (Djones.Lettnin@infineon.com)

Infineon Technologies AG, München, Germany

Abstract— **The proposed work aims to develop a Core-Monitor for Hardware-Software co-debugging targeting emulation platforms. Core-Monitor is required to have a general hardware and software co-debugging environment for an embedded processor. It generates a trace file that monitors the runtime behavior of the CPU. A processor trace file would record the events in the CPU and render the core instructions in a human-readable form while it requires a post-processing tool for a meaningful interpretation of the data in these trace files. The trace file contents typically include the program counters, instructions, memory read/write operations, register access data, and their executed time. These trace files are provided as input to tools like Indago for post-processing and debugging.**

Keywords—HW/SW Co-Debugging, Emulation, Trace file, ZeBu, SV-DPI, Indago

I. Introduction

Debugging and verification are two of the most significant challenges in modern System-on-Chip (SoC) design. Current SoCs are composed of numerous processing engines, various peripherals, and hierarchical network-on-chip architectures. Therefore, debugging techniques are indispensable to developing and operating multicore SoCs. Debugging is unpredictable and varies significantly from project to project; hence co-debugging of hardware and software substantially contributes to time to market delays for new products. In order to ensure productivity and keep up with the increasing complexity of hardware and software co-design, it is essential to develop new techniques and methodologies for hardware and software co-debugging.

Intending to reduce the HW/SW co-debugging time, there is now a race to process data faster using less power, which probes further challenges. The developers need to trace interactions across multiple and often heterogeneous processing elements that have to function independently of each other. However, some prominent processor providers developed their own tracing capabilities; for instance, ARM provides a tarmac trace feature [1] a standard for ARM cores.

When multiple cores are put together in a system, there is no reusable solution to trace the processor state of these cores. For processors like OpenRISC, RISCV, and others, the favored method for generating the trace file is to build a module for each processor, generally written in Verilog or SystemVerilog, which contains several lines of code that are needed to trace the processor state. This method is feasible for small SoC designs with one or two processors. Embedded processors are developed as multicore implementations. It means that we have to build up many modules for every processor, which leads to redundancy as each module contains the same lines of code to trace the processor state and hence is effort and time-consuming.

Emulation is now a critical component of advanced chip design verification and is evolving to meet future demands for increasingly dense, complex, and heterogeneous architectures. Emulation is a process that converts a design into an implementation capable of being executed on a specialized hardware [2]. With the increasing complexity of designs, simulation is no longer a desirable approach for HW/SW co-debugging. Larger designs have moved towards emulation-based HW/SW co-debugging techniques. However, due to the complexity of the emulator architecture, implementing HW/SW co-debugging techniques in the emulation platform is more critical than in the simulation. Emulators impose various tradeoffs on cost, performance, and turnaround time. As a result, transitioning from simulators to emulators for a design can be quite challenging. There is no reusable platform that can be used in both the simulation and emulation environments for HW/SW co-debugging.

The Core-Monitor aims to solve this problem by being a standard for trace file generation in simulation and emulation environments, which reduces the effort to create the processor state tracer, reduces the HW/SW co-debugging time, and shortens the time to market.

The rest of the paper is structured as follows: Section II gives a brief overview of the related work and background in hardware software co-debugging techniques. Section III discusses the architecture of the Core-Monitor setup and proposes a methodology used for the implementation of the Core-Monitor. Section IV illustrates the Core-Monitor setup in the emulation environment with the help of a use case. Then it discusses the evaluation metrics considered and provides a summary of the relevant results obtained in this work. Finally, Section V provides a conclusion of this paper along with an outlook for future work.

II. BACKGROUND AND RELATED WORK

A. *Related Work*

While there are many debug, simulation, emulation, and verification tools available, none are acclaimed as the best since each offers more convenient features in certain circumstances. As a result, the engineer must manage some of these tools and select the appropriate one depending on the stage of the project or the nature of the failure. ARM CoreSight™ [3] is an ARM-developed on-chip component that enables multicore cross triggering by allowing a core to reach a breakpoint and disable the other cores. Using ARM CoreSight, [4] presents a hardware architecture for monitoring memory operations and combining data transfer addresses and data transfer values in real-time. ARM CoreSight technology creates a unique debug trace stream standard for memory monitoring without interfering with the regular operation of the system. ARM developed the tarmac trace utility [5] for ARM products such as Fast Models and Cycle Models, as well as direct simulations of CPUs from its RTL to run in a mode that generates a detailed trace of a program's execution. These traces are typically written to a text file in the *tarmac* format. *tarmac* is a textual format that logs the instructions and their effects executed by a CPU. It maintains a list of every value written to a register, every value read and written from memory, and other events like interrupts and exceptions. However, this methodology is unique to ARM cores and cannot be reused.

Table 1: Summary of Related Work; f - fully implemented; p - partially implemented

Related Work	Summary	Multi-Core	Trace file	Re-usable
[6]	Proposes an assertion-based debugging to deal with HW/SW concurrency issues at the system level using virtual platforms	f		
[7], [8]	An open-source tool is proposed that makes use of GNU debugger (GDB) instance to a full-system simulation of an embedded FPGA system in ModelSim	p	p	
[9]	Proposes a concept of a multi-level re-targetable debugger for ASIPs	p	p	p
[10], [11]	Approach for multicore HW/SW co-debugging in multiple instruction multiple data (MIMD) type design using ARM CoreSight.	f	f	
[12]	ARM Coresight-based real-time hardware tracing solutions are proposed in this work.	p	f	

The above Table 1 summarizes the state-of-the-art research in Hardware-Software co-debugging. The column *Multi-Core* means that the proposed technique supports processors with multiple cores. The column *Trace file* is to indicate if the proposed technique generates the trace file for HW/SW co-debugging, and the column *Re-usable* demonstrates that the proposed methodology is re-usable for various processors. As illustrated in the table, while the proposed techniques support multicore approaches, little research has been conducted on their reusability for SoCs with multiple cores. There is currently no reusable platform for emulation and simulation environments, which is addressed in this work.

B. Simulation vs Emulation

HDL simulators are prevalent as the primary tool for design verification in the electronic engineering community. They are highly adaptable to debugging a hardware design since they are simple to use and quick to set up and compile a design. They may be able to accommodate numerous design iterations each day, provided that the design size is suitable. They get challenging for tens of millions of gate counts, typical for today's system-level design verification. Since the modern SoCs have grown more extensive and complex in terms of gate counts and supported features, speed of the simulators have become a primary bottle neck. This leads to alarming consequences about the completeness of verification by limiting the number of simulation tests to meet the demands of the market [13].

Table 2: Comparison of Simulation versus Emulation Based on Performance, Design Capacity, and Setup/Compile Time

	Simulator	**Emulator**
Performance	in kHz	in MHz
Design Capacity	upto few million gates	upto billion gates
Setup and Compile time	minutes to days	days to weeks

On the other hand, hardware emulation can detect practically all design flaws, whether in the hardware or embedded software of the SoC. They can handle large design sizes but need considerable setup time and are slow to compile compared to the simulator. Simulators can compile a design in a few minutes while the emulators take hours to complete the compilation task. Table 2 indicates the threshold of performance, design capacity and the time taken to setup and compile a design in simulators and emulators. It illustrates how the emulator compromises with setup and compilation time while increasing performance and capacity. The performance of the HDL simulators can go up to few kHz and they drop significantly as the design capacity exceeds few million gates. Since emulators are running on actual clock speed and each of its hardware components are running at mega-hertz speed, throughput of a design becomes really high and this reduces the overall verification cycle [14]. As a result, more complex design projects have migrated to the emulation platform for verification and debugging.

C. ZeBu and Direct Programming Interface (DPI)

The Synopsys emulator ZeBu is the emulator used in this work. Synopsys ZeBu [15] is an extremely high-capacity emulation system designed to facilitate SoC verification requirements for automotive, 5G, networking, artificial intelligence, and more. ZeBu supports DPI imported SystemVerilog function calls within the DUT [16]. The SystemVerilog Direct Programming Interface (DPI) is an interface between SystemVerilog and other programming languages. It enables the designer to easily invoke C functions from SystemVerilog and export SystemVerilog functions for use in C. Unlike transactor calls, the ZeBu hardware-software infrastructure that supports these calls allows communication exclusively from hardware to software (ZeBu to Host) to enhance runtime efficiency. Hence, this work uses the DPI interface and serves as a transactor for the CPU state tracer in the BFM.

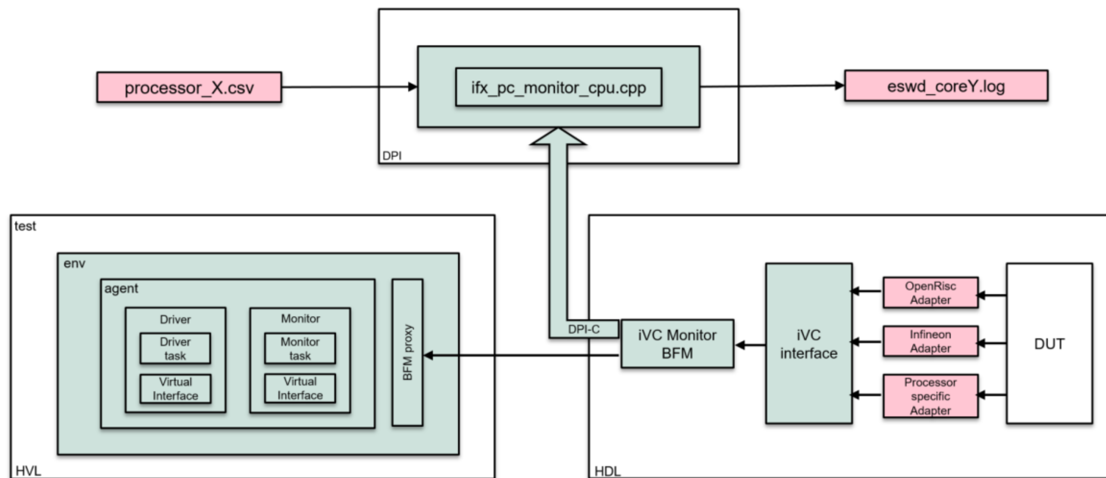
III. METHODOLOGY AND IMPLEMENTATION

This work aims to develop a Core-Monitor for Hardware-Software co-debugging targeting emulation platforms. Core-Monitor is required to have a general hardware and software co-debugging environment for an embedded processor. It generates a trace file that monitors the runtime behavior of the CPU. A processor trace file would record the events in the CPU and render the core instructions in a human-readable form while it requires a post-processing tool for a meaningful interpretation of the data in these trace files. The trace file contents typically include the program counters, instructions, memory read/write operations, register access data, and their executed time. These trace files are provided as input to tools like Indago [2] for post-processing and debugging.

A. Architecture and Methodology

The architecture of the Core-Monitor is depicted in Fig 1. It comprises of processor adapter modules that are CPU-dependent; iVC (interface Verification Component) interface, BFM (Bus Functional Module) module and the DPI block that are considered as CPU-independent. CPU-dependent means that the functionality is specific to the processor in consideration and is color-coded in red whereas CPU-independent means that the functionality is generic to all the processors that use the Core-Monitor and is color-coded in green. The methodology proposed in this work consists of a CPU dependent adapter to perform the initial processing of received processor signals from processor hardware implementation. The adapter is specific to the processor and is the only module that has to be updated by the user for different processors. The adapter is connected to the rest of the Core-Monitor components that are CPU-independent. These include an iVC interface and iVC BFM for the timed hardware synthesis, a DPI-C component that provides essential processor tracing functions, and the state-of-the-art verification and acceleration technology, SystemVerilog UVM for the untimed verification domain. This UVM testbench framework remains as a black box throughout the work.

Fig 1: Architecture of Core-Monitor



B. Implementation

The testbench is partitioned into two tops, *hdl_top* and *hvl_top* which helps to enable the testbench to be run in the co-emulation setup [17]. The synthesizable components are instantiated in the *hdl_top* whereas the untimed behavioral components that includes the UVM testbench are in the *hvl_top*. This is shown in Fig 1. The synthesized *hdl_top* runs on the emulator and *hvl_top* runs on the simulator. The HDL and HVL communicate at the transaction level. This communication is enabled by using a SystemVerilog virtual interface. Core-Monitor also implements a DPI-C interface that serves as a transactor for the CPU state tracer in the BFM.

The adapter is implemented as a timed HDL module rather than an untimed SystemVerilog UVM class type in the HVL domain. It is responsible for the initial processing of processor signals received from the DUT due to the uncertainty regarding the conditions under which to capture the processor state signals. This uncertainty is primarily because different processors have varying hardware implementations, resulting in a range of conditions for when to receive the processor status signal. Additionally, it acknowledges that CPUs with similar cores may have varying implementations due to multiple approaches followed by the manufacturer and their design styles. As a result, the conditions under which the needed processor status signal is processed and collected will differ.

The generated signals will be committed to the iVC interface following the initial processing. The user must manually write and instantiate the adapter for every processor. The benefit of this method is that the user can construct a single adapter for a SoC with multiple cores if all of the cores use the same processor hardware design. This is accomplished by instantiating the adapter multiple times, depending on the number of similar cores. Thus, the adapter can be reused for similar cores in the processor.

```
interface ifx_pc_monitor_if#(parameter CPU_PC_NUM = 16,
                             parameter CPU_REG_NUM = 100)
                             (input logic [7:0] cpu_num);
 // Signals
 logic cpu_clk;
 logic cpu_reset_n;
 logic [ifx_pc_monitor_pkg::MAX_PC_LENGTH-1:0] cpu_pc[CPU_PC_NUM];
 // program counter array
 logic [ifx_pc_monitor_pkg::MAX_INSTRUCTION_LENGTH-1:0]
 cpu_instruction[CPU_PC_NUM];  // executed instruction array
 logic cpu_pc_valid[CPU_PC_NUM];
 // undefined instruction (particular PC)
 logic [ifx_pc_monitor_pkg::MAX_REG_DATA_LENGTH-1:0]
 reg_data[CPU_REG_NUM];     // register array
 logic [ifx_pc_monitor_pkg::MAX_MEM_ADDR_LENGTH-1:0] cpu_mem_addr;
 // memory store/load address
 logic [ifx_pc_monitor_pkg::MAX_MEM_DATA_LENGTH-1:0] cpu_mem_data;
 // memory load/store data
 logic [ifx_pc_monitor_pkg::MAX_MEM_SIZE-1:0] cpu_mem_size;
 // memory size
 logic cpu_mem_ack;
 //acknowledgement from the memory subsystem
 logic cpu_mem_str_enable;
 // memory write enable
 logic [ifx_pc_monitor_pkg::MAX_CPU_LENGTH-1:0] cpu_id;
 // id for the processor config file
```

Fig 2: CPU-Independent iVC Interface

The iVC interface is a SystemVerilog interface comprised of standard signals that are recognized by all the processor adapters. Hence it is a CPU-independent component. It serves as a connection between the CPU-dependent adapter and the CPU-independent iVC BFM. As illustrated in Fig 2, these universal signals include the following, clock signal, *cpu_clk* and the reset signal, *cpu_reset_n*. The signals for PC tracing, i.e., *cpu_pc*, *cpu_instruction* and *cpu_pc_valid*. It includes the signal, *reg_data* for register tracing. The signals, *cpu_mem_addr*, *cpu_mem_data*, *cpu_mem_size*, *cpu_mem_ack*, and *cpu_mem_str_enable* are used for memory tracing. Additionally, it features a signal *cpu_id*, which is helpful in getting processor-specific variables from the configuration file in the DPI-C function. The PC count and register count are parameterized in the interface definition, allowing them to be overwritten during the instantiation of the interface module. Additionally, the interface consists of a parameter *cpu_num* that specifies the active core of the processor, which is helpful in instantiating the interface for multiple cores of the same processor. The interface must be declared as a passive modport to allow the Core-Monitor to monitor all essential signals.

The iVC monitor BFM is responsible for collecting the processed signals from the iVC interface. Simultaneously, it imports the C function calls via the DPI interface and passes them to the C functions responsible for dumping the processor state signals into the corresponding trace files. Moreover, this component is CPU independent and does not require any modification from the user.

The SystemVerilog Direct Programming Interface (DPI) is a powerful tool for integrating models written in other languages with Verilog designs. This is intended to improve runtime performance and reusability by utilizing a higher-level language. The compilation and build phase, which is a substantial time sink in the emulation setup, is ramped up with the use of the DPI block. The DPI-C block comprises a .cpp file and a header file containing the necessary functions for creating trace files for all processor cores and writing the appropriate signals from the iVC BFM via the DPI-C interface. These processor specific trace files are depicted as *eswd_coreY.log* in Fig 1, where *Y* points to the respective core that is traced. Moreover, it provides simple user interaction as it enables direct access to processor-specific variables in the C file via a configuration file. This is named as *processor_X.csv* in Fig 1. Typically, the configuration file contains variables that the user must modify before the iVC is instantiated for the same or various processors in use. For instance, this comprises the number of cores in the processor, the number of registers, and the names of the register to be traced in the trace file. Table 3 shows an instance of the configuration file for a processor with 3 cores and 5 registers. The changes to these configuration variables are made at the top level. Hence neither the iVC BFM nor the iVC interface needs to be modified by the user. As a result, the DPI-C component is likewise regarded as CPU-independent. Additionally, it is compatible with simulation and emulation systems.

Table 3: Configuration File of a Processor

CPU_CORES	3				
CPU_CORE_NAMES	core1	core2	core3		
CPU_REGS	5				
CPU_REG_NAMES	R0	R1	R2	R3	R4
CPU_INSTRUCTION	1				

The top module must instantiate an n-core processor n times. Then, at the top level of the testbench, the iVC instances must be configured with the appropriate processor mode and connected to the respective processor. This methodology presents a reusable and CPU-independent iVC for processor state tracing. It is necessary to use one or more adapters to perform the initial processing of the processor signals. Hence, the Core-Monitor is a unified approach for trace file generation in simulation and emulation environments.

IV. Results

The Core- Monitor is tested with OpenRISC and RISCV processors in the simulation environment and the IFX processor in the emulation environment. This paper presents a use-case for the emulation system using the Synopsys emulator ZeBu in the following section.

A. *Use Case*

Infineon Technologies AG provided a SoC project (here briefly called IFX project for confidentiality) to support the development of the Core-Monitor in the emulation environment. Synopsys emulator ZeBu is used for this purpose. The IFX processor had nine cores with five active cores, and here the adapter was instantiated five times in the connect module with the appropriate signals from the DUT to the adapter signals. The minimum code required to implement such an adapter in SystemVerilog is approximately 290 lines. This is core-specific and varies accordingly. Then the iVC interface and the IFX processor adapter are instantiated and integrated into the IFX project. Furthermore, the iVC BFM is instantiated with the iVC interface. After completing the Core-Monitor design, the emulation environment has to be set up using the emulator ZeBu. Firstly, an RTL testbench variant is selected that is provided by the developers of the IFX processor. After the desired testbench variant has been selected, resource libraries are compiled. Then, the time-consuming build phase of the ZeBu emulator is started. ZeBu supports the build phase in the batch and GUI modes. This build operation will compile the HDL part of the testbench and the HVL part, including the DPI. This process will approximately take 5 hours for the IFX processor design. Once the compilation is complete, the design can be downloaded to an emulator for emulating the design. The emulation requires the memory image of the testbench in .sre format. Fig 3 shows the output of the trace file generated for one of the IFX processor cores. For instance, in line 6 the register A10 of core0 in the IFX processor is written to 0xd0009820 (hexadecimal format) at 7176200ps.

```
1  5826200 ps PC=0xaffff f00
2  7126200 ps PC=0xafffc000
3  7176200 ps PC=0xafffc004
4  7176200 ps REG=A10, 0xd0010000
5  7226200 ps PC=0xafffc008
6  7226200 ps REG=A10, 0xd0009820
7  7276200 ps PC=0xafffc00c
8  7276200 ps REG=A15, 0xf0060000
9  7326200 ps REG=A15, 0xf0064404
10 8026200 ps PC=0xafffc010
11 8076200 ps REG=D15, 0x00010001
12 8126200 ps PC=0xafffc012
13 8176200 ps MR=0xf0064404, 0x00010001
14 8476200 ps PC=0xafffc016
15 8476200 ps PC=0xafffc01a
16 8526200 ps REG=A15, 0xf0060000
17 8576200 ps REG=D15, 0x44140000
18 9626200 ps PC=0xafffc01e
19 9676200 ps PC=0xafffc022
20 9676200 ps PC=0xafffc026
21 9676200 ps REG=A15, 0xf0064410
22 9726200 ps PC=0xafffc028
23 9776200 ps PC=0xafffc02c
24 9776200 ps REG=D15, 0x44141224
25 9776200 ps REG=A15, 0xf0060000
```

Fig 3: Trace file obtained for IFX processor

The generated trace file is used for post-processing and debugging of the processor design, it is an input to post-processing HW/SW co-debugging tools such as Indago that generate debug databases. After preparing the debugging databases, Indago ESWD is launched to debug embedded software. The source code, assembly code,

and waveform with recorded PC and C functions can be synchronized for debugging purposes. Additionally, the user can incorporate hardware signals into the waveform to conduct co-debugging of hardware and software.

B. Performance Evaluation

Table 4: Emulation Runtime

Emulation Runtime	Core-Monitor	Existing Method
Testcase 1 (with eswd)	3197.3s	3635.9s
Testcase 1 (without eswd)	160.3s	239.8s
Testcase 2 (with eswd)	889.2s	921.3s
Testcase 2 (without eswd)	34.4s	52.7s

The performance comparison between the existing setup for emulation and the Core Monitor setup is compared and evaluated. The evaluation metric considered for this purpose is emulation runtime. It is the total time taken by the test case to complete the emulation. This time is reported at the end of the complete emulation run of the test case and is recorded in seconds. Two test cases, namely, *Testcase 1* and *Testcase 2*, are evaluated using *eswd* [18] option and without using *eswd* option during the emulation run. The *eswd* option is necessary for trace file generation. The average emulation runtime values for ten runs are recorded in Table 4 for each test case. *Testcase 1* is a long-running test case compared to *Testcase 2*. For instance, the emulation runtime in the Core- Monitor setup for *Testcase 1* executed with the *eswd* option is 53.3 minutes (3197.3 seconds) while it is 60.6 minutes (3635.9s) in the existing setup. Similarly, *Testcase 1* takes 2.7 minutes (160.3 seconds) and 4 minutes (239.8s) in the Core-Monitor and the Existing setup respectively when executed without the *eswd* option. Table 4 shows that the emulation run with the *eswd* option consumes more time as it will generate trace files with enormous amounts of data that impact the overall performance. Hence, developing a Core-Monitor that does not significantly deteriorate the performance is critical. It is seen that the Core-Monitor is slightly faster than the existing emulation setup for trace file generation. Nonetheless, this is an added advantage as the Core-Monitor setup is user-friendly and only requires the user to design a processor-specific adapter, unlike the existing setup where the entire structure of the monitor was mainly for a specific processor and was not re-usable.

V. CONCLUSION

This paper proposes a novel methodology that targets a reusable Core-Monitor to monitor CPU signals for the different kinds of processors. Additionally, this methodology is compatible with being used in simulation and emulation platforms. The experimental results demonstrate that the proposed methodology is reusable for the different kinds of processors. This is achieved as the user must design only the adapter for the processor. The proposed methodology does not require any change in the Core-Monitor components when switching from emulation to simulation environment and vice-versa. This is enabled as the Core-Monitor is developed so that it can be synthesized in the emulator and in the simulator environments. The experimental evaluation in the two environments shows that the new methodology is better and does not significantly deteriorate the performance compared to the already existing methods of CPU tracing in the respective platforms. As the SoC design size is considerably increasing, the Core-Monitor approach decreases the design of the HW/SW co-debugging technique for each processor. It substantially reduces the coding effort of redundant modules and hence increases reusability. Furthermore, the developed processor state tracer is designed to be integrated into the automatic testbench framework, adding to the reusability factor.

Since the proposed novel methodology of the Core-Monitor for hardware and software co-debugging is highly user-friendly, it is reusable for all kinds of processors. This enables the future developers to integrate many other processor types into the currently implemented processor state tracer. The future developers can devise a versatile approach capable of parsing the user configuration of the processor in both C functions and SV modules.

DESIGN AND VERIFICATION™ 2022 DVCON CONFERENCE AND EXHIBITION EUROPE MUNICH, GERMANY DECEMBER 6 - 7, 2022

REFERENCES

[1] ARM Developer, "TarmacTrace," ARM, [Online]. Available: https://developer.arm.com/documentation/100964/1118/Plug-ins-for-Fast-Models/TarmacTrace. [Accessed March 2022].

[2] S. Engineering, "Emulation," [Online]. Available: https://semiengineering.com/knowledge_centers/eda-design/verification/emulation/. [Accessed March 2022].

[3] ARM Developer, "Introducing coreSight: debug and trace infrastructure," [Online]. Available: https://developer.arm.com/documentation/102520/latest/. [Accessed March 2022].

[4] S. M. A. Zeinolabedin, J. Partzsch and C. Mayr, "Analyzing ARM coreSight ETMv4.x data trace stream with a real-time hardware accelerator," in *2021 Design, Automation & Test in Europe Conference & Exhibition*, 2021.

[5] ARM Community, "Tarmac Trace Utilities," [Online]. Available: https://community.arm.com/arm-community-blogs/b/tools-software-ides-blog/posts/tarmac-trace-utilities. [Accessed August 2022].

[6] L. G. Murillo, R. L. Bücs, D. Hincapie, R. Leupers and G. Ascheid, "SWAT: assertion-based debugging of concurrency issues at system level," in *The 20th Asia and South Pacific Design Automation Conference*, 2015.

[7] R. Willenberg and P. Chow, "Simulation-based HW/SW co-debugging for field-programmable systems-on-chip," in *2013 23rd International Conference on Field programmable Logic and Applications*, 2013.

[8] T.-H. Chang, S.-C. Hou and . I.-J. Huang, "A unified GDB-based source-transaction level SW/HW co-debugging," in *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, 2016.

[9] J. Křoustek, Z. Přikryl, K. Dušan and H. Tomáš, "Retargetable multi-level debugging in HW/SW codesign," in *ICM 2011 Proceeding*, 2011.

[10] B. Vermeulen, R. Kühnis, J. Rearick, N. Stollon and G. Swoboda, "Overview of debug standardization activities," *IEEE Design & Test of Computers,* vol. 25, no. 10.1109/MDT.2008.81, pp. 258-267, 2008.

[11] K.-J. Lee, A. Su, L.-F. Chen, J.-W. Jhou, J. Kuo and M. Liu, "A software/hardware co-debug platform for multi-core systems," in *2011 9th IEEE International Conference on ASIC*, 2011.

[12] S. M. A. Zeinolabedin, J. Partzsch and C. Mayr, "Real-time hardware implementation of ARM coresight trace decoder," in *IEEE Design Test*, 2021.

[13] V. Billa and S. Haran, "Challenges and mitigations of porting a UVM testbench from simulation to transaction-based acceleration (co-emulation)," in *DVCon United States*, 2018.

[14] J. Aggarwal, T. Nguyen and P. K. Gupta, "DFT+emulation – a faster way to close verification," in *IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2018.

[15] Synopsys, "Emulation," [Online]. Available: https://www.synopsys.com/verification/emulation.html.

[16] Synopsys, "ZeBu Server User Guide," [Online]. Available: https://www.synopsys.com/support.html. [Accessed November 2021].

[17] H. Sharma, H. v. d. Schoot and A. Murarka, "Unifying hardware-assisted verification and validation using UVM and emulation," in *DVCon United States*, 2013.

[18] Cadence, "Indago Embedded Software Debug," Cadence, [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/software-driven-verification/indago-embedded-sw-debug-app.html. [Accessed 2022].