

**AN EMBEDDED TEST PLATFORM FOR SYSTEM-ON-A-CHIP INTERFACES**

**WILMER DANIEL RAMIREZ VERA**

**UNIVERSIDAD INDUSTRIAL DE SANTANDER  
FACULTAD DE INGENIERÍAS FISICOMECÁNICAS  
ESCUELA DE INGENIERÍA ELÉCTRICA, ELECTRÓNICA Y DE  
TELECOMUNICACIONES  
BUCARAMANGA  
2019**

**AN EMBEDDED TEST PLATFORM FOR SYSTEM-ON-A-CHIP INTERFACES**

**WILMER DANIEL RAMIREZ VERA**

**Trabajo de grado presentado como requisito para optar al título de  
Magister en Ingeniería de Telecomunicaciones**

**Director**

**ÉLKIM FELIPE ROA FUENTES  
INGENIERO ELECTRICISTA. PhD.**

**Codirector**

**CKRISTIAN RICARDO ESTEBAN DURÁN BLANCO  
INGENIERO ELECTRÓNICO. MSc.**

**UNIVERSIDAD INDUSTRIAL DE SANTANDER  
FACULTAD DE INGENIERÍAS FISICOMECÁNICAS  
ESCUELA DE INGENIERÍA ELÉCTRICA, ELECTRÓNICA Y DE  
TELECOMUNICACIONES  
BUCARAMANGA  
2019**

## **Acknowledgments**

In this section, all the people and institutions involved in the realization of the microcontroller where the debugging platform is placed will be mentioned.

- Elkim Roa  
Direction of the work and director of the integrated systems research group Onchip. Research group where this project is developed. Contributions in the revision and edition of this document
- Ckristian Durán.  
Co-direction of the work. In charge of the sign-off process of the microcontroller, and digital design assistant.
- Hector Gómez.  
Mixed-signal adaptation of the microcontroller. Contributions in the revision and edition of this document.
- Juan Romero.  
Digital designer and IP integration assistant in the microcontroller project. In charge of system buses connection.
- Universidad Industrial de Santander (UIS) and Integrated Systems Research Group Onchip  
For providing the software tools and equipment necessary for the execution of this project.
- All Onchip team that was part of the digital and analog design of the microcontroller project.

## CONTENT

|                                                                                     |           |
|-------------------------------------------------------------------------------------|-----------|
| <b>INTRODUCTION</b>                                                                 | <b>15</b> |
| <b>1 ENABLED-SOC DEBUG PLATFORM</b>                                                 | <b>21</b> |
| 1.1 GENERAL DESCRIPTION OF THE PLATFORM . . . . .                                   | 22        |
| 1.2 JTAG TEST-ACCESS-PORT AND BOUNDARY-SCAN . . . . .                               | 24        |
| 1.2.1 Test-Access-Port . . . . .                                                    | 25        |
| 1.2.2 Boundary-Scan Register . . . . .                                              | 28        |
| 1.3 DEBUG MODULE . . . . .                                                          | 31        |
| 1.4 SYSTEM BUS MONITOR . . . . .                                                    | 35        |
| <b>2 HDL SIMULATION USING THE UNIVERSAL VERIFICATION METHOD-<br/>OLOGY</b>          | <b>40</b> |
| 2.1 THE UNIVERSAL VERIFICATION METHODOLOGY . . . . .                                | 40        |
| 2.1.1 UVM structure for verification IP . . . . .                                   | 41        |
| 2.2 A UVM-BASED VIP TO TEST THE DEBUGGING PLATFORM HDL<br>DESCRIPTION VIP . . . . . | 43        |
| 2.3 SIMULATION RESULTS . . . . .                                                    | 45        |
| <b>3 SOFTWARE INTERFACE FOR DEBUGGING</b>                                           | <b>53</b> |
| 3.1 SOFTWARE FOR DEBUGGING . . . . .                                                | 53        |
| 3.1.1 The GNU debugger . . . . .                                                    | 54        |
| 3.1.2 Open SoC Debug . . . . .                                                      | 56        |
| 3.1.3 Open On-Chip Debugger . . . . .                                               | 56        |
| 3.2 IMPLEMENTED SOFTWARE INTERFACE . . . . .                                        | 59        |
| <b>4 IMPLEMENTATION RESULTS</b>                                                     | <b>66</b> |
| 4.1 SYNTHESIS RESULTS . . . . .                                                     | 66        |
| 4.2 EXPERIMENTAL RESULTS . . . . .                                                  | 68        |
| 4.2.1 Set-up for experimental measurements . . . . .                                | 70        |
| 4.2.2 Planning and measurement results . . . . .                                    | 72        |
| 4.2.3 Experimental results and discussion . . . . .                                 | 78        |
| 4.3 FPGA IMPLEMENTATION RESULTS . . . . .                                           | 80        |
| 4.4 SUMMARY . . . . .                                                               | 88        |



|                                        |            |
|----------------------------------------|------------|
| <b>5 CONTRIBUTIONS AND FUTURE WORK</b> | <b>90</b>  |
| 5.1 CONTRIBUTIONS . . . . .            | 90         |
| 5.1.1 Related articles . . . . .       | 90         |
| 5.1.2 Bachelor's Projects . . . . .    | 91         |
| 5.2 FUTURE WORK . . . . .              | 92         |
| <b>REFERENCES</b>                      | <b>94</b>  |
| <b>BIBLIOGRAPHY</b>                    | <b>104</b> |

## LIST OF TABLES

|    |                                                                                                                                          |    |
|----|------------------------------------------------------------------------------------------------------------------------------------------|----|
| 1  | Instructions supported by the implemented JTAG TAP & BS. . . . .                                                                         | 26 |
| 2  | Composition of DM status data register. . . . .                                                                                          | 28 |
| 3  | Description of AHB protocol signals (1). . . . .                                                                                         | 32 |
| 4  | Selection of the comparison type performed by the filter. The comparison is done with <i>Reference</i> and <i>Offset</i> values. . . . . | 37 |
| 5  | Association of <i>Reference</i> and <i>Offset</i> registers to the AHB signals. . . . .                                                  | 37 |
| 6  | Sequences implemented in the UVM-based VIP for the debugging platform. . . . .                                                           | 46 |
| 7  | UVM results of debugging platform verification. . . . .                                                                                  | 52 |
| 8  | Resources Comparison with Other Debugging Systems. . . . .                                                                               | 67 |
| 9  | Planning of the experimental tests. . . . .                                                                                              | 73 |
| 10 | Approval indicators of the tests. . . . .                                                                                                | 73 |
| 11 | Key Features of the proposed debugging platform. . . . .                                                                                 | 88 |

## LIST OF FIGURES

|    |                                                                                                                                                                   |    |
|----|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 1  | Life cycle of system-on-chip development. . . . .                                                                                                                 | 15 |
| 2  | Common problems of the testing stage and their consequences. . .                                                                                                  | 16 |
| 3  | Simplified block diagram of debugging platform connection in an SoC. .                                                                                            | 18 |
| 4  | Summary of the proposal of this research work. . . . .                                                                                                            | 19 |
| 5  | Simplified block diagram of the SoC to integrate with the debugging platform. . . . .                                                                             | 21 |
| 6  | Simplified block diagram of the proposed debugging platform connected to the SoC. . . . .                                                                         | 23 |
| 7  | Simplified block diagram of the test-access-port logic (2). . . . .                                                                                               | 26 |
| 8  | Structure of the DM_Access data register. . . . .                                                                                                                 | 27 |
| 9  | Structure of the DM status data register. . . . .                                                                                                                 | 27 |
| 10 | TAP controller state diagram (2). . . . .                                                                                                                         | 29 |
| 11 | Boundary-scan cell digital circuit (2). . . . .                                                                                                                   | 30 |
| 12 | Simplified block diagram of boundary-scan register connected to general purpose IOs. The boundary-scan design is scalable and can be adapted for $n$ IOs. . . . . | 30 |
| 13 | Simplified block diagram of the debug module. . . . .                                                                                                             | 31 |
| 14 | Operation illustration of the core control provided by DM registers .                                                                                             | 33 |
| 15 | Behavior illustration of the AHB Bus interface operated by DM registers. . . . .                                                                                  | 35 |
| 16 | Simplified version of the filter stage in the bus monitor. . . . .                                                                                                | 36 |
| 17 | Simplified version of the performance analysis stage in the BM. . .                                                                                               | 38 |
| 18 | Top view of the simplified block diagram of the System bus monitor. .                                                                                             | 39 |
| 19 | Conceptual illustration of the UVM logic for VIP. . . . .                                                                                                         | 41 |
| 20 | Flowchart of the execution steps of a UVM-based VIP. . . . .                                                                                                      | 42 |
| 21 | Top view of the UVM structure for VIP. . . . .                                                                                                                    | 43 |
| 22 | Top view of UVM-based VIP for the debugging platform. . . . .                                                                                                     | 44 |
| 23 | Conceptual illustration of the UVM-Based VIP process for the debugging system. . . . .                                                                            | 45 |
| 24 | Simulated waveforms of halt core operation of the debug module. .                                                                                                 | 47 |
| 25 | Simulated waveforms of a single loading instruction to a program buffer cell. . . . .                                                                             | 47 |

|    |                                                                                                                                                                                                            |    |
|----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 26 | Simulated waveforms of storing instructions in program buffer cells and running these. . . . .                                                                                                             | 47 |
| 27 | Simulated waveforms of AHB transactions using the debug module. . . . .                                                                                                                                    | 48 |
| 28 | Simulated waveforms of putting GPIO_0 high by using AHB transfers. . . . .                                                                                                                                 | 48 |
| 29 | Simulated waveforms of an AHB reading transaction. . . . .                                                                                                                                                 | 49 |
| 30 | Simulated waveforms of reading transaction to the DM register for AHB response. . . . .                                                                                                                    | 49 |
| 31 | Simulated waveforms of monitor operation with filter config: Comparison:IDsrc, Operation: range, Reference: 100 offset: 50, Latency range W0: $\leq 5$ , Latency range W1: [6, 10]. . . . .                | 51 |
| 32 | Simulated waveforms of Monitor operation with filter config: Comparison:IDsrc, Operation: AND, Reference: 80 (slave id) Offset: 384(Data). Latency range W0: $\leq 5$ , Latency range W1: [6, 10]. . . . . | 51 |
| 33 | Simplified diagram block of the hardware-software communication. . . . .                                                                                                                                   | 53 |
| 34 | Example of debugging of a dual-core system using OSD. . . . .                                                                                                                                              | 57 |
| 35 | Top-level architecture of OpenOCD. . . . .                                                                                                                                                                 | 58 |
| 36 | Example of the debug connection using OpenOCD (simplified version). . . . .                                                                                                                                | 59 |
| 37 | Illustration of software-hardware connection. . . . .                                                                                                                                                      | 59 |
| 38 | Conceptual diagram of the composition of files set-up for the software interface. . . . .                                                                                                                  | 60 |
| 39 | Composition of C libraries and functions to control the software interface. . . . .                                                                                                                        | 62 |
| 40 | Representation of the C functions to operate the debugging platform. . . . .                                                                                                                               | 63 |
| 41 | Code examples of programming the debugging platform. . . . .                                                                                                                                               | 64 |
| 42 | Debugging platform synthesized into a 32-bit microcontroller. . . . .                                                                                                                                      | 68 |
| 43 | Micrograph of the 32-bit microcontroller silicon implementation. . . . .                                                                                                                                   | 69 |
| 44 | Test boards for the 32-bit microcontroller chip. . . . .                                                                                                                                                   | 70 |
| 45 | Illustration of the connection set-up to test the microcontroller. . . . .                                                                                                                                 | 72 |
| 46 | IOs control by the boundary-scan register. . . . .                                                                                                                                                         | 74 |
| 47 | Photographs of the board in the execution of the blinking generated by boundary-scan. . . . .                                                                                                              | 75 |
| 48 | Simplified block diagram of the JTAG structure. A deepening in the data registers module is presented showing where the error is detected. . . . .                                                         | 76 |
| 49 | Code description of the device initialization. . . . .                                                                                                                                                     | 81 |
| 50 | Measured waveforms of the initialization of the device. These signals are captured by the ILA. . . . .                                                                                                     | 81 |
| 51 | Measured waveforms of the data sending to the DM. . . . .                                                                                                                                                  | 82 |
| 52 | Information displayed in Linux console. . . . .                                                                                                                                                            | 82 |
| 57 | Measured waveforms of the AHB system bus transaction. . . . .                                                                                                                                              | 82 |

|    |                                                                              |    |
|----|------------------------------------------------------------------------------|----|
| 53 | Code example to send data to the DM. . . . .                                 | 83 |
| 54 | Measured waveforms of the DM writing: halt and reset orders. . . .           | 83 |
| 55 | Program execution displayed in Linux console for a DM writing. . .           | 84 |
| 56 | Code example of an AHB system bus transaction. . . . .                       | 84 |
| 58 | Program execution about AHB transactions displayed in Linux console. . . . . | 85 |
| 59 | Debug module status printed in Linux console. . . . .                        | 85 |
| 60 | Code example of using the function <i>DM_info()</i> . . . . .                | 86 |
| 61 | Execution of the program in Figure 60. . . . .                               | 87 |

## Resumen

**TÍTULO:** Plataforma embebida de pruebas para interfaces de sistemas en chip. \*

**AUTOR:** WILMER DANIEL RAMIREZ VERA †

**PALABRAS CLAVES:** Depuración, sistemas en chip, pruebas post silicio.

Los sistemas de depuración para sistemas en chip (*SoC* por sus siglas en inglés) deben ofrecer capacidad de control de ejecución y visibilidad de complejos *SoCs* con el fin de detectar y/o analizar errores y encontrar posibles mejoras de diseño. La ausencia de estos sistemas puede conducir a una pérdida de tiempo y dinero debido a fallas que no se pueden detectar o corregir. Entonces, la necesidad de un sistema que permita realizar pruebas en silicio crece junto con la complejidad actual de los *SoC*. Este trabajo presenta una plataforma de depuración escalable y reutilizable para pruebas posteriores al silicio. La plataforma está compuesta por un módulo de depuración comunicado a través de JTAG, y un monitor de bus. Las características relevantes son el control del procesador, operación del bus del sistema, y el monitoreo no intrusivo de las operaciones en el *SoC*. Un filtrado flexible permite seleccionar transferencias de interés o realizar un monitoreo general del *SoC*. Los datos capturados se pueden analizar mediante contadores de rendimiento que verifican las transferencias ejecutadas y finalizadas, calculando su latencia para detectar puntos muertos en el sistema.

---

\*Trabajo de Investigación.

†Facultad de Ingenierías Fisicomecánicas. Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones. Director: Élkim Felipe Roa Fuentes. Codirector: Ckristian Ricardo Esteban Durán Blanco.

La plataforma de depuración se ha implementado como parte de un *SoC* con un procesador de 32 bits basado en el set de instrucciones RISC-V, y múltiples periféricos. Además, este trabajo presenta una interfaz de software para realizar las diferentes operaciones de la plataforma.

## Abstract

**TITLE:** An Embedded Test Platform for System-on-a-Chip Interfaces <sup>‡</sup>

**AUTHOR:** WILMER DANIEL RAMIREZ VERA <sup>§</sup>

**KEYWORDS:** Debug, system-on-a-chip, post-silicon testing

Post-silicon debugging systems must offer run-control capability and visibility on complex systems-on-a-chip (SoC) in order to detect/analyze errors and find possible design enhancements. The absence of these systems can lead to waste of time and money due to failures that can not be detected or corrected. Then, the need for a system that allows testing on silicon grows along with the current complexity of SoCs. This work presents a scalable and reusable debugging platform for post-silicon testing. The platform is composed of a debug module with JTAG communication and a bus monitor. Relevant features are core control, system bus operation, and non-intrusive monitoring. A flexible filtering allows selecting transfers of interest or performing general monitoring of the SoC. Captured data can be analyzed by performance counters that check executed and finished transfers, calculating their latency to detect deadlocks in the system. The debugging platform has been implemented as part of an SoC with a 32-bit RISC-V based core and multiple peripherals. In addition, this work presents a software interface to perform the different operations of the platform.

---

<sup>‡</sup> Research Work

<sup>§</sup> Facultad de Ingenierías Fisicomecánicas. Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones. Advisor: Élkim Felipe Roa Fuentes. Coadvisor: Ckristian Ricardo Esteban Duran Blanco.



## INTRODUCTION

Integrated circuits (IC) research has allowed the development of smaller electronic systems with better performance in speed and energy consumption. ICs have been developed to support operational blocks working together, with the capability of processing instructions and executing processes. These ICs are known as systems-on-chip (SoCs). Figure 1 shows a common commercial time development flow of an SoC (3). Figure 1 displays how in most of the flow steps, verification and testing steps are performed. In the pre-silicon stage (before fabrication), multiple simulations must be performed to test analog and digital circuits in different stages of the process. Moreover, simulations of the full system are performed to check its behavior. Verification techniques and methodologies have allowed creating robust testbenches to verify all the possible functionalities of SoC designs (4; 5; 6; 7; 8; 9; 10). However, the verification process does not finish in the pre-silicon stage. The fabricated SoC must pass through a testing stage before going to market.

The testing stage involves injecting signals into the SoCs in silicon, and extracting data about their behavior. Some processes commonly executed for test-

Figure 1. Life cycle of system-on-chip development.

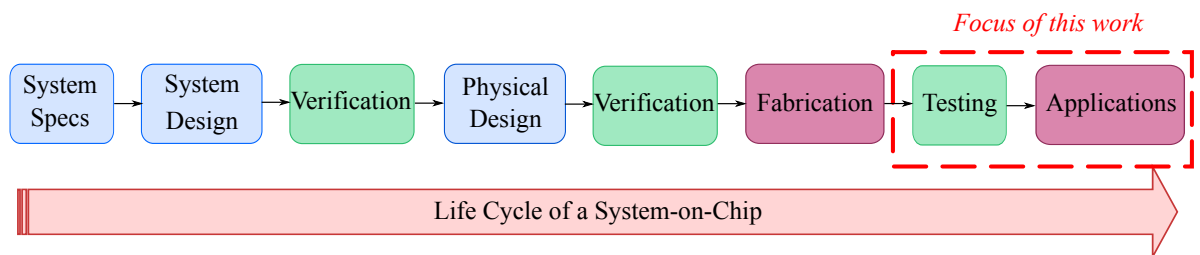
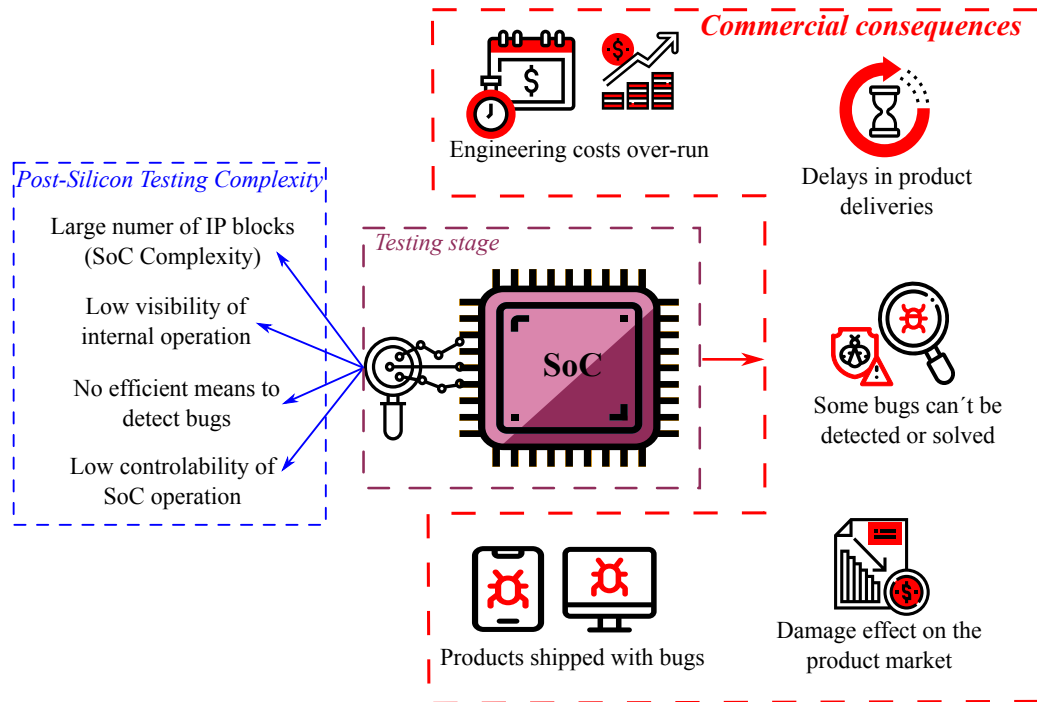


Figure 2. Common problems of the testing stage and their consequences.



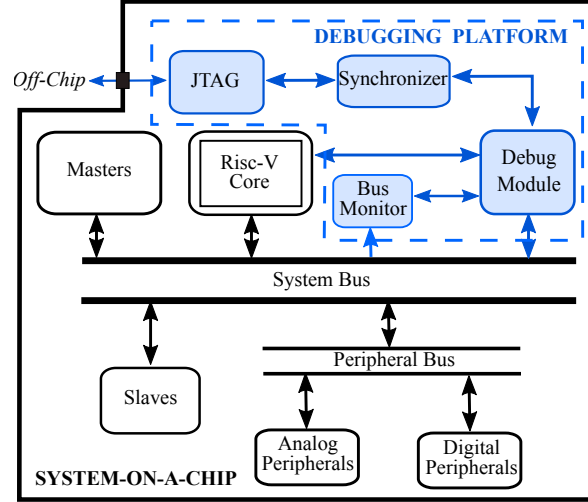
ing are: applying tests to the different circuits, performing operation and performance analysis, and taking measurements of SoC circuits. These processes help to detect bugs in the system, differences with the pre-silicon verification, or to find design factors that can be improved in future versions (11). However, the testing stage depends directly on the complexity of the SoC.

SoC devices have become increasingly complex, causing that the testing process involves a greater effort that is represented in time and money. Even, in many cases, SoC complexity makes the testing may not cover all the analysis that should be performed. For instance, Figure 2 presents significant commercial consequences that can appear: delay in products (potentially affecting their success on the market); engineering costs over-run; products are shipped to customers with lingering bugs that were simply too difficult to fix them. Moreover, subtle

problems occur in the field with no efficient means to detect them or correct them (12). To address these problems, on-chip embedded systems for debugging are incorporated into the design of the SoC. These systems help to monitor its internal processes, capture signals and exercise some control in the SoC operation. The advantages of these systems have allowed to develop reliable electronic devices, and to detect and solve multiple errors in silicon, thanks to the visibility and controllability that they offer in the SoC. Although the implementation of debugging systems can consume up to one-third of the design effort required to create a complex SoC (13), these systems are required for the reliability of the product.

SoC requires debugging systems that can offer run-control, visibility, and flexibility to successfully test its operation in a post-silicon testing (14; 15). The run-control is necessary to perform operations within the SoC that are later analyzed to detect bugs. Visibility is desired to observe the systems that are part of the SoC, and flexibility is a key feature to perform detailed analyzes in different tests. Figure 3 shows a simplified SoC block diagram with a debugging system highlighted in blue. Run-control capability allows the debugging system to perform halt/injected operations in the SoC. By software, different operations can be forced to be executed through the debugging platform. In the platform, the run-control capability is provided by controlling the core and performing system bus transactions. On the other hand, visibility is associated with the capacity of monitoring transactions between subsystems. The bus monitor provides visibility of the SoC by sampling the different transactions performed through the system bus, which is responsible for interconnecting SoC circuits. Finally, capturing data is not enough to perform an efficient analysis. The SoC analysis must allow making different execution cases within the SoC. Therefore, a configurable data filtering is employed to provide this flexibility in the SoC analysis.

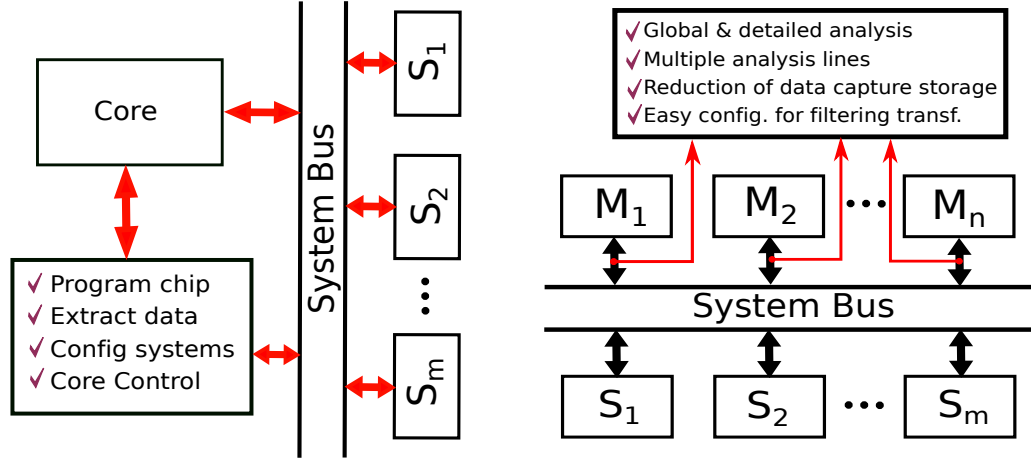
Figure 3. Simplified block diagram of debugging platform connection in an SoC.



Reported debugging systems usually focus on finding errors or abnormalities in specific processes of an SoC. In (15; 16), a shared debugging unit monitors transactions of masters connected to the system bus, in order to find errors as deadlocks, livelocks or data inconsistencies. However, the limited flexibility of this system reduces the possibility to monitor particular transactions. In (17), a design-for-debug (DfD) employs trace buffers and trigger units to test functions. Although this system achieves real-time observability and avoids excessive storage on-chip, the system is limited to certain operation cases. Kyung et al. in (18) present a bus monitor with the capability to configure address ranges, which is not enough to perform detailed analysis.

This thesis proposes a system focused on improving the flexibility and coverage of post-silicon validation analysis. A debugging method based on a central debug module (DM) with JTAG communication and a bus monitor (BM) is presented. The proposed system provides configurable coverage levels to perform post-silicon validation, and supports filter configuration to detect system bus transactions. Figure 4 illustrates the features of the system proposed in this work.

Figure 4. Summary of the proposal of this research work.



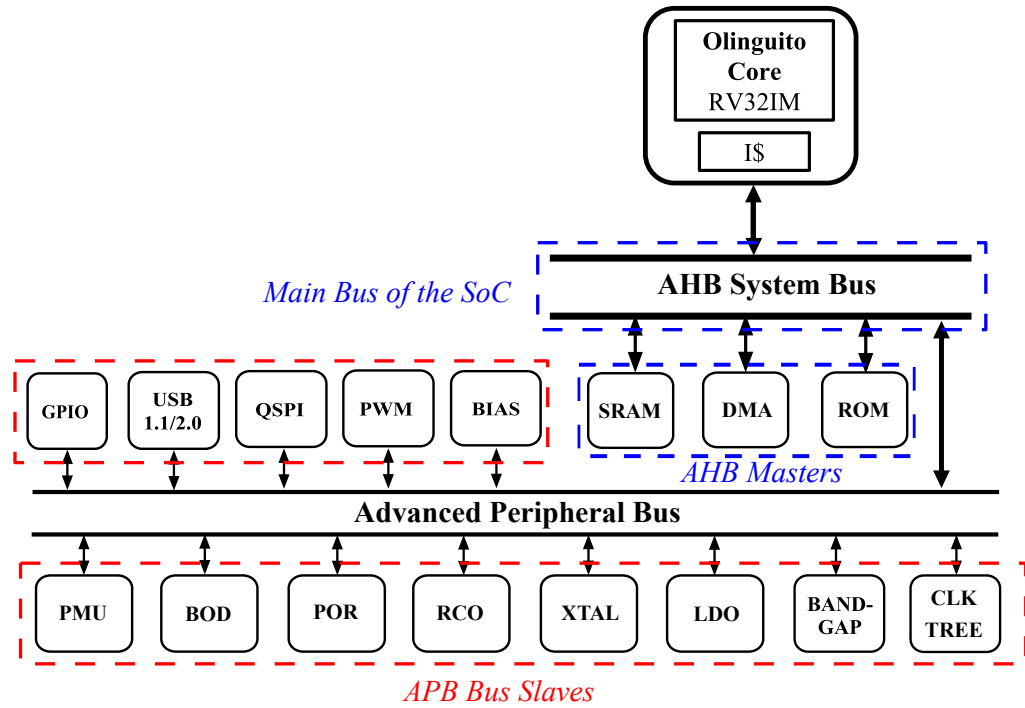
The design is based on a scalable and reusable system. Run-control and monitoring systems with the features indicated in Figure 4 are described. In addition, a software interface that controls and executes the functions supported by the debugging platform is presented.

In order to expose the proposed research work, this thesis starts by describing the design of the debugging platform in Chapter 2. It presents the three parts of the platform: JTAG, the debug module and the bus monitor, by explaining their structure and operation. To test the design and check its correct operation, we require a functional verification process. Thus, Chapter 3 presents the pre-silicon verification process performed for the debugging platform. Chapter 3 initiates with a section focused on the theory of the Universal Verification Methodology (UVM), which is employed in functional verification for SoCs. Later in the chapter, the methodology is applied by describing a testbench to test the proposed design. To finish Chapter 3 exposes the simulation results. Once the pre-silicon verification has been presented, the thesis describes the software component that will allow controlling the designed platform. To expose the software-level research work,

Chapter 4 shows software techniques applied in the industry for debug applications, and describes the implemented software interface for the proposed platform. Chapter 5 highlights how the designed hardware and software are tested working together with the presentation of the platform results. Synthesis results of the debugging platform integrated in a 32-bit microcontroller are reported. Experimental tests are applied to a silicon implementation of the microcontroller that includes a part of the platform. In the implementation, an error is detected and corrected. Then, the full microcontroller is implemented in an FPGA to test the functions of the enhanced platform. The software interface is employed to operate the microcontroller implemented in the FPGA. To visualize the processes that occur inside the FPGA, internal logical analyzers (ILA) are used to observe the signals associated with the operations of the debugger. Finally, Chapter 6 presents the contributions of this thesis and future work.

## 1. ENABLED-SOC DEBUG PLATFORM

Figure 5. Simplified block diagram of the SoC to integrate with the debugging platform.



This chapter presents the description of the proposed debugging platform and its operation within an SoC. Figure 5 depicts a simplified diagram block of an SoC which corresponds to a microcontroller. This microcontroller is mainly composed of a 32-bit core, two interconnection systems called buses, and the masters and slaves of these buses. The buses implemented in this microcontroller are AMBA, the main bus is AHB (another option is AXI) and the secondary bus is APB (1; 19). The shown microcontroller does not have a debugging system in its architecture. The purpose then, is to integrate the debugging system with this microcontroller.

The designed platform allows off-chip communication in order to receive instructions and data in the microcontroller, and extract information from this. Off-

chip communication is achieved by implementing the JTAG protocol (2). A central debug module is proposed to manage the run-control of the debugging operations executed on the 32-bit core and the system buses. Moreover, the debug module provides visibility of the SoC's peripherals to access them. Finally, a monitoring system is designed to observe and analyze the transactions performed on the system buses. Architectures and logic schemes of the subsystems that compose the platform are presented.

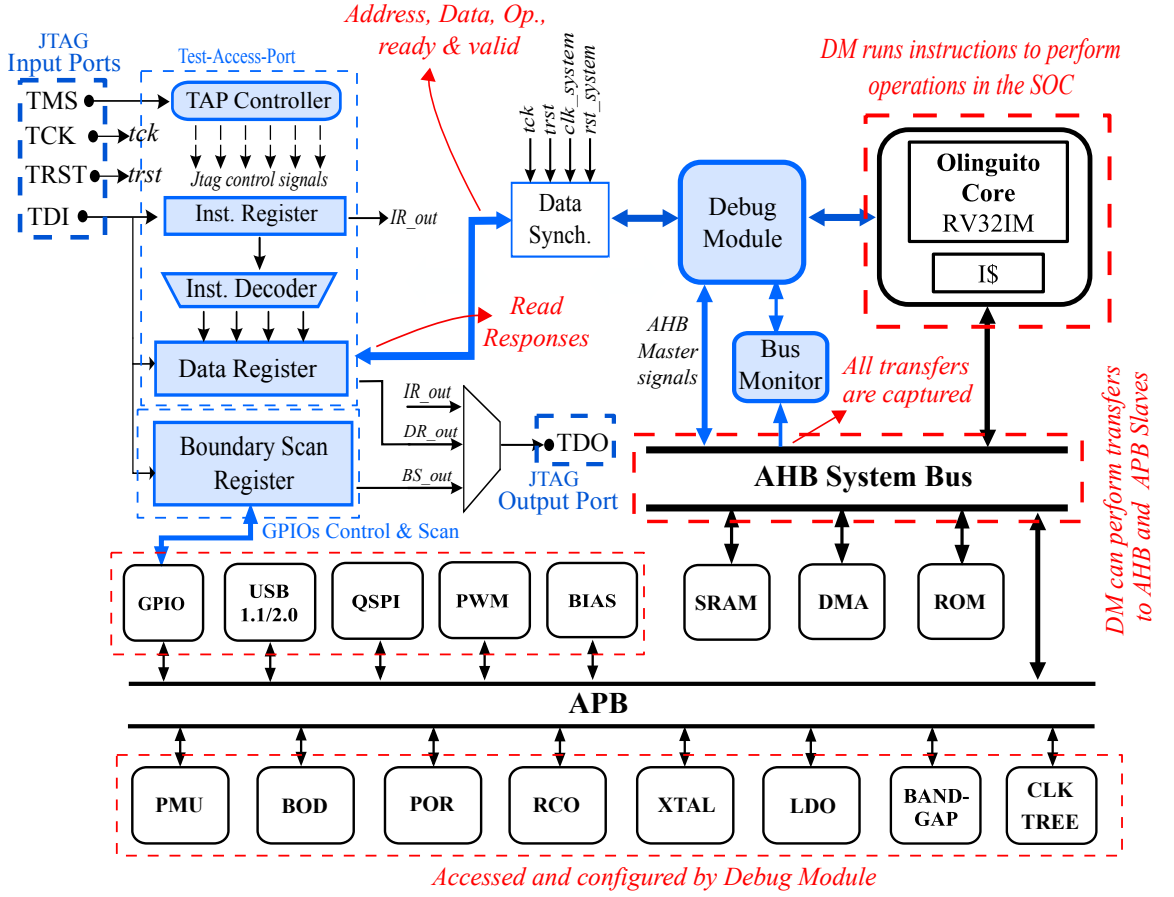
The debugging system is described using Chisel 3.0 which is an open-source HDL embedded in the Scala programming language and developed at the University of California in Berkeley (20). Chisel supports advanced hardware design using highly parameterized generators and layered domain-specific hardware languages. The design described using Chisel can be compiled and synthesized to construct a Verilog file of this design. Then, the design described in Verilog is tested using a verification methodology presented in Chapter 3. The application of the verification methodology to test the platform design and the simulation results are also presented in Chapter 3.

## **1.1. GENERAL DESCRIPTION OF THE PLATFORM**

Figure 6 presents a simplified general block diagram of the proposed debugging platform along with a block diagram of the SoC. The proposed debugging platform consists of three main parts: JTAG, central debug module, and bus monitor. JTAG (Joint Test Action Group) is a serial protocol that allows off-chip communication. This is divided into two parts: the test-access-port (TAP) and the boundary-scan which are designed based on IEEE Std 1149.1-2013 (2). The TAP is responsible for receiving data and instructions serially, and performing actions associated with



Figure 6. Simplified block diagram of the proposed debugging platform connected to the SoC.



that information. On the other hand, the boundary scan is a system for control and observation of the SoC GPIOs. In order to communicate with the DM, additional fields were added to the instruction and data registers in the TAP, as in (21). Debug instructions are serially sent through JTAG port, deserialized and stored at the data register .

The DM is the central controller of the debugging platform. This controller contains all configuration registers that control debugging operations such as: halting the core, resetting the core, and activating the debug mode so that the

core runs instructions that are provided by the DM. A register field called program buffer stores the core instructions. Some registers reserved for the BM are also contained in the DM to configure filters and to perform BM counting. The register field supports transactions based on the AHB protocol, enabling communication with any slave connected to the system bus (AHB) or the peripheral bus (APB). In addition, this DM structure is adaptable for any other AMBA bus protocol as AXI (19).

The proposed system bus monitor on Figure 6 provides a flexible configuration that allows performing either detailed or general analysis. The BM performs a non-intrusive operation by observing and capturing transfers without intervening in the operation of the system bus. The captured transfers pass through filters that operate based on the configuration of control registers. The filter passes the selected data to counters modules. The BM is scalable in the number of filters and performance counters. Each filter associates a performance counter, and each filter has an independent configuration register, thereby offering the possibility to perform multiple independent analyses.

## **1.2. JTAG TEST-ACCESS-PORT AND BOUNDARY-SCAN**

JTAG is an integrated technology for testing systems and interconnections at the integrated circuit (IC) level. Since its introduction as an industry standard, JTAG has become a protocol commonly implemented not only in system-level verification but also as a development debugging tool (22). JTAG protocol is based on IEEE Std 1149.1-2013 (2). This protocol enables access to pins and on-chip infrastructure IP. Thus, JTAG helps in the testing process in silicon stage, and reduce time spent in verification and the costs that this implies (23). IEEE 1149.1/JTAG can be divided into two subsystems: 1) test-access-port and 2)

boundary-scan register. This section presents the composition and the features of these subsystems.

**1.2.1. Test-Access-Port** The test-access-port (TAP) is a general-purpose port that can provide access to many test support functions built into a debugging component, including the test logic defined by the standard in (2). The TAP can be implemented using at least three input ports and one output port. A fourth input port can be included to operate as reset. Figure 7 illustrates a conceptual schematic of the TAP logic. The TAP ports are defined as follows:

- **Test-Data-Input (TDI):** Serial input that receives test instructions and data.
- **Test-Clock-Input (TCK):** Provides the clock for the TAP.
- **Test-Mode-Select (TMS):** This signal is sampled in a rising edge at TCK. TMS value determines the next state of the TAP controller, the state machine that controls test operations.
- **Test-Data-Output (TDO):** Serial output for test instructions and data.
- **Test-Reset-Input (TRST):** Reset signal for the TAP.

The instruction register stores the instructions supported by the JTAG design. Table 1 describes the supported instructions in the proposed design. Each instruction has a unique binary code which is used to decode the instruction once TDI data are deserialized. Then, the respective data register is selected to receive and store data through TDI. The first four instructions in Table 1 are mandatory by IEEE Std 1149.1-2013. This design includes two additional instructions to transfer instructions and data to the DM (DM\_Access) and obtain information about its operation status (DM\_Status).

Figure 7. Simplified block diagram of the test-access-port logic (2).

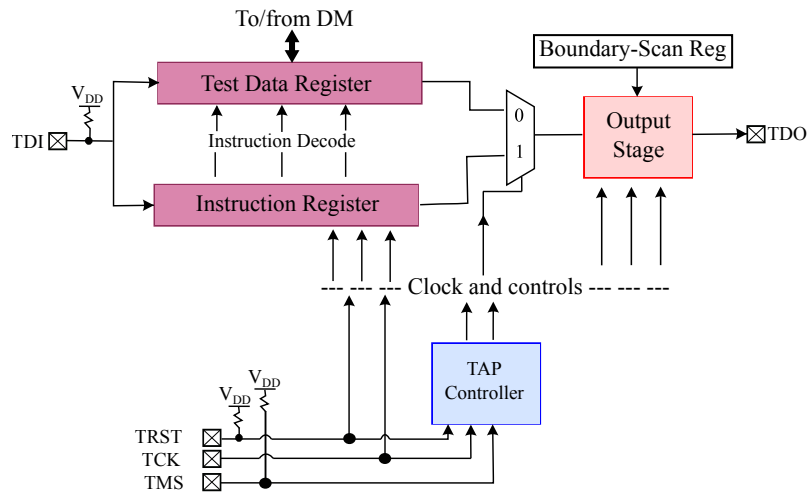


Table 1. Instructions supported by the implemented JTAG TAP & BS.

| Name           | Code | Description                                                                                      |
|----------------|------|--------------------------------------------------------------------------------------------------|
| Bypass         | 1111 | Selects the bypass register to be connected for serial access between TDI and TDO                |
| IDCode         | 0010 | Selects the device identification register to be connected for serial access between TDI and TDO |
| Extest         | 0000 | Allows testing of off-chip circuitry and board-level interconnections. (for Boundary-Scan)       |
| Sample Preload | 0001 | Snapshot of GPIOs activity (Sample) and load data in Boundary-Scan registers (Preload)           |
| DM.Access      | 1000 | Allows sending instructions and data to the Debug Module through DM data register                |
| DM.Status      | 1001 | Access to the DM.Status register that contains a general status of the DM operation              |

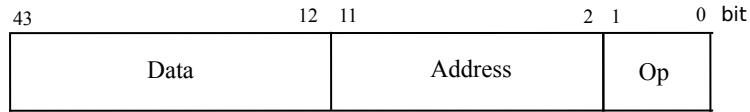


Figure 8. Structure of the DM\_Access data register.

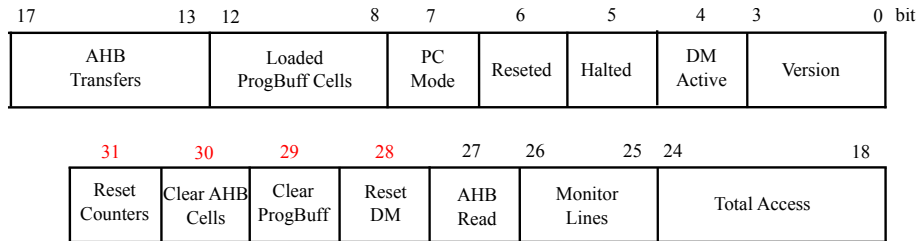


Figure 9. Structure of the DM status data register.

Figures 8 and 9 depict the data registers for the last two instructions of Table 1. The DM Access data register contains three signals: *operation*, *address*, and *data*. *Operation* indicates if a write or read operation is performed, the *address* indicates the address of the DM register to be accessed, and *data* is the data to be written in the selected register. The DM status data register has only four bits for writing, which in Figure 9 are indicated in red. Bits 0-27 are connected to read-only registers. Table 2 summarizes each of the data stored in the de DM status data register.

Figure 10 shows the diagram state of the TAP controller. The diagram state shows adjacent to each state transition the value at TMS in a rising edge at TCK. TAP controller is only controlled by TMS. Figure 10 depicts a data column and an instruction column. This separation between the states allows the loading of instructions and data in a simple way. The instruction column must be executed first to select the instruction to be performed, and subsequently, the appropriated data register loads data in the states of the data column. The data transfer occurs specifically in the Shift-DR and Shift-IR states. The instruction and the data to be operated are only updated in Update-IR and Update-DR respectively. After these

Table 2. Composition of DM status data register.

| Name                   | Type | Description                                        |
|------------------------|------|----------------------------------------------------|
| Version                | R    | Stores the version of the DM                       |
| DM Active              | R    | Indicates if the DM is activated                   |
| Halted                 | R    | Indicates if the core is halted                    |
| Reseted                | R    | Indicates if the core is reseted                   |
| PC Mode                | R    | Indicates if the PC mode in DM is activated        |
| Loaded Prog-Buff Cells | R    | Indicates how many program buffer cells are loaded |
| AHB Transfers          | R    | Number of performed AHB transactions               |
| Total Access           | R    | Total number of access to the DM                   |
| Monitor Lines          | R    | Activated monitor lines                            |
| AHB Read               | R    | Indicates if there is an AHB response              |
| Reset DM               | W    | Resets all Registers in the DM                     |
| Clear ProgBuff         | W    | Clears all program buffer cells                    |
| Clear AHB cells        | W    | Clears all AHB cells                               |
| Reset counters         | W    | Resets the status counters in the DM               |

states, the operations to be performed with this data can be executed.

**1.2.2. Boundary-Scan Register** The boundary-scan register works as a shift register that allows testing external circuitry, by sampling IOs activity in a non-intrusive operation. The boundary-scan register contains cells that allow sampling the state of an IO that operates as an input, or injecting a value for an IO that operates as an output. Boundary-scan cell circuit is shown in Figure 11. The implemented boundary-scan is connected to the IOs system. These IOs have six signals: input, output, input enable, output enable, pull up, and ds. Each signal associates a boundary-scan cell to be controlled or sampled. Figure 12 illustrates the boundary-scan register implemented. The shift register captures the IOs values and shifts them to TDO. Also, the shift register can configure the six control signal for the IOs by shifting values injected by TDI. To perform the boundary-scan actions, the instruction register must be first loaded with an instruction associated

Figure 10. TAP controller state diagram (2).

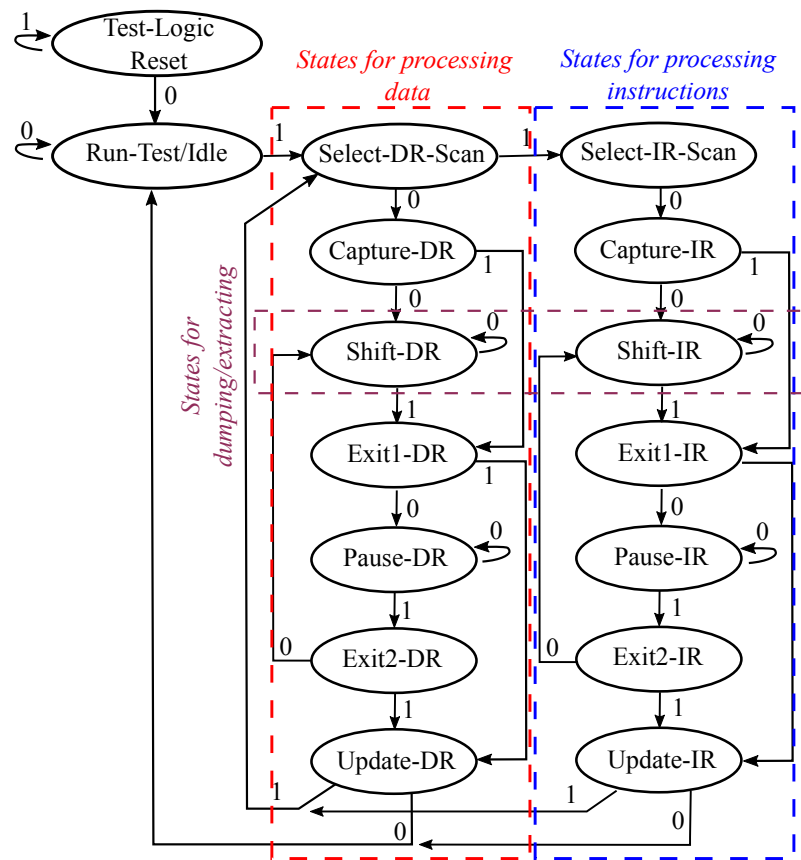


Figure 11. Boundary-scan cell digital circuit (2).

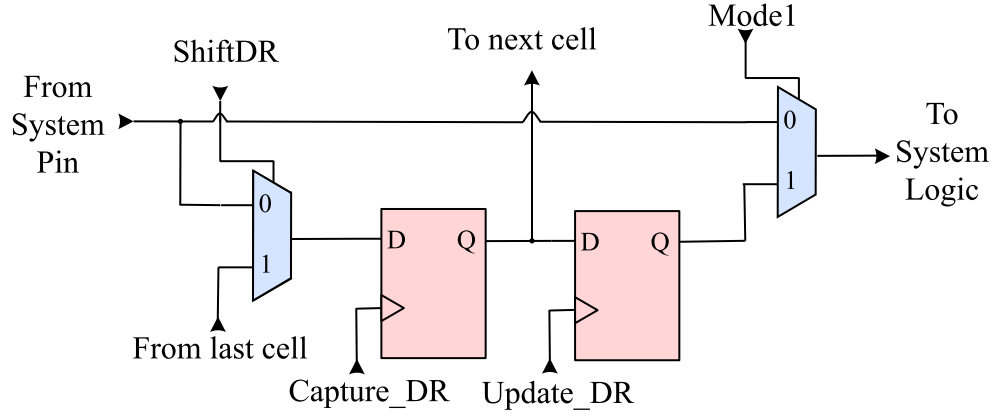
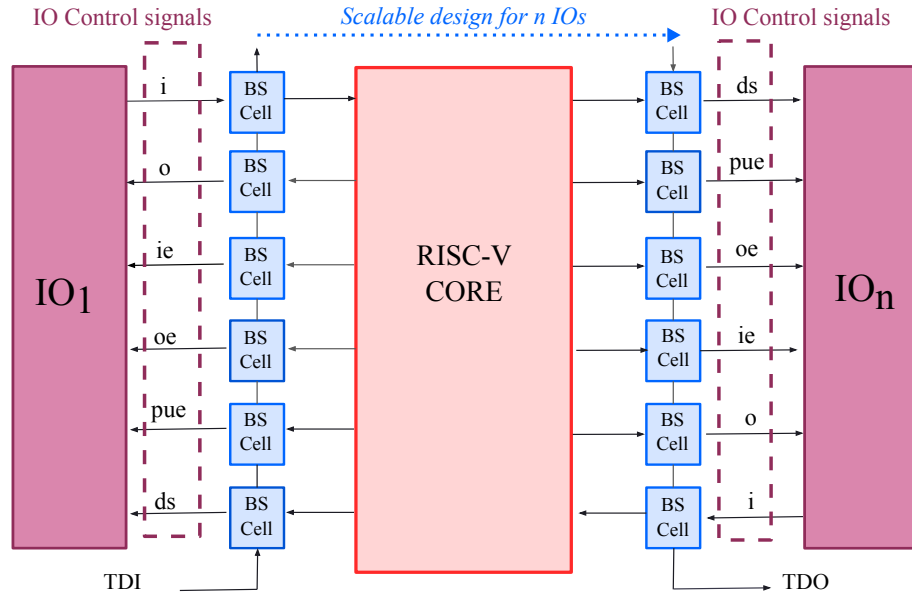


Figure 12. Simplified block diagram of boundary-scan register connected to general purpose IOs. The boundary-scan design is scalable and can be adapted for  $n$  IOs.



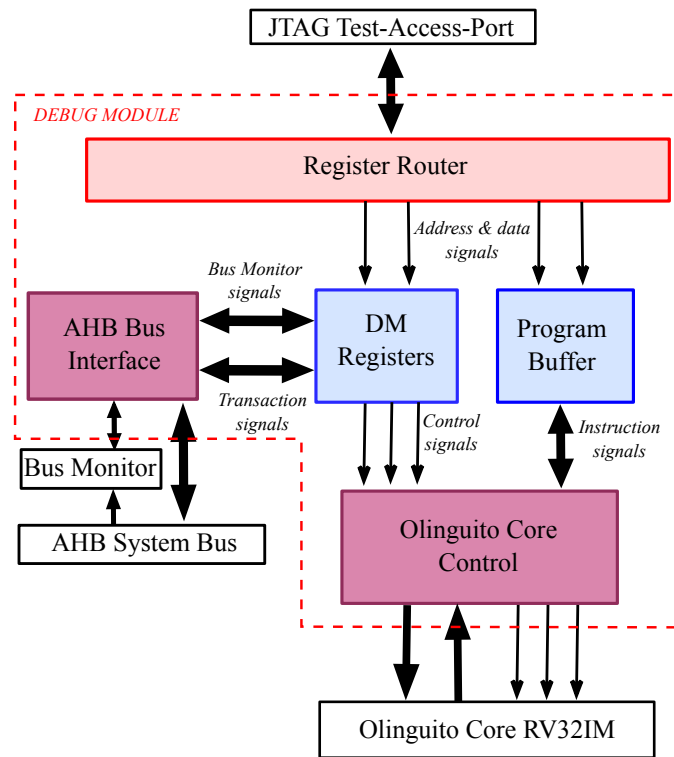
with the boundary-scan. Table 1 describes two of these instructions: *Sample\_Preload* and *Extest*. In *Sample\_Preload*, the *Mode1* signal value in Figure 11 corresponds to zero (for a non-intrusive operation). On the other hand, in *Extest*, the *Mode1* signal value corresponds to one (inject values to the IOs).



### 1.3. DEBUG MODULE

The debug module is the central system in the proposed debugging platform. In this module, all debugging instructions are performed. The DM associates a register map that stores the configuration of the debugging operation. A register router (RR) controls the reading or writing of the register map. The RR receives three buses from the TAP, containing: 32-bit data, a 32-bit address, a 2-bit operation bus, along with signals to synchronize a data transfer with the TAP (Figure 6). The RR handles routing transactions to the appropriate register. For a writing operation, the RR overwrites the register value. For a reading operation, the RR sends the response data through another data channel addressed to the TAP.

Figure 13. Simplified block diagram of the debug module.



Register map of the DM adjusts debugging operations and controls them.

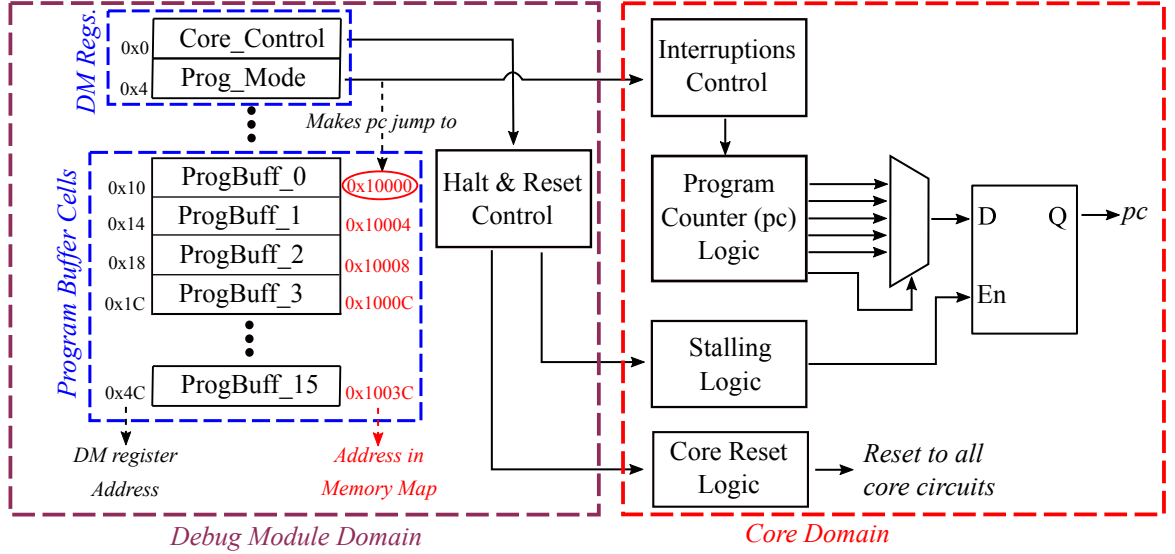
Figure 13 presents a block diagram of the proposed DM structure. Two registers groups can be observed: DM registers that contain all configuration-and-execution registers for debugging, and the program buffer which stores instructions that can be executed by the core. DM registers configuration allows halting and resetting the core, as well as activating a program mode. The program mode makes the program counter (pc) of the core to jump to a dedicated memory space for the program buffer. The design of the program buffer is scalable and the number of registers can be easily modified. Additionally, the program buffer is designed as a memory space with bidirectional communication with the core. In other words, the program buffer can be also written by the core.

Table 3. Description of AHB protocol signals (1).

| Signal   | Length | Description                                 |
|----------|--------|---------------------------------------------|
| Haddress | [31:0] | Address of the slave connected to the bus   |
| Hwdata   | [31:0] | Data to be transferred in a write operation |
| Hwrite   | [0:0]  | Indicates: Writing (high) or Reading (low). |
| Htrans   | [1:0]  | Indicates the transfer type                 |
| Hsize    | [2:0]  | Indicates the size of the transfe           |
| Hready   | [0:0]  | Confirms the end of a operation             |
| Hrdata   | [31:0] | Data returned in a reading operation        |

Figure 14 depicts a block diagram about the connection between the DM and the core, for the core control process. Figure 14 shows some debug module registers including the program buffer cells. The number of these cells can be configured to synthesize an SoC with a minimum of 4 and a maximum of 16 program buffer cells. Each DM register has a specific address in the DM, and the program buffer cells additionally have an address associated with the memory map of the SoC. Therefore, these cells can be accessed by another system bus master. The first memory address of the program buffer (0x10000) is used in a

Figure 14. Operation illustration of the core control provided by DM registers .

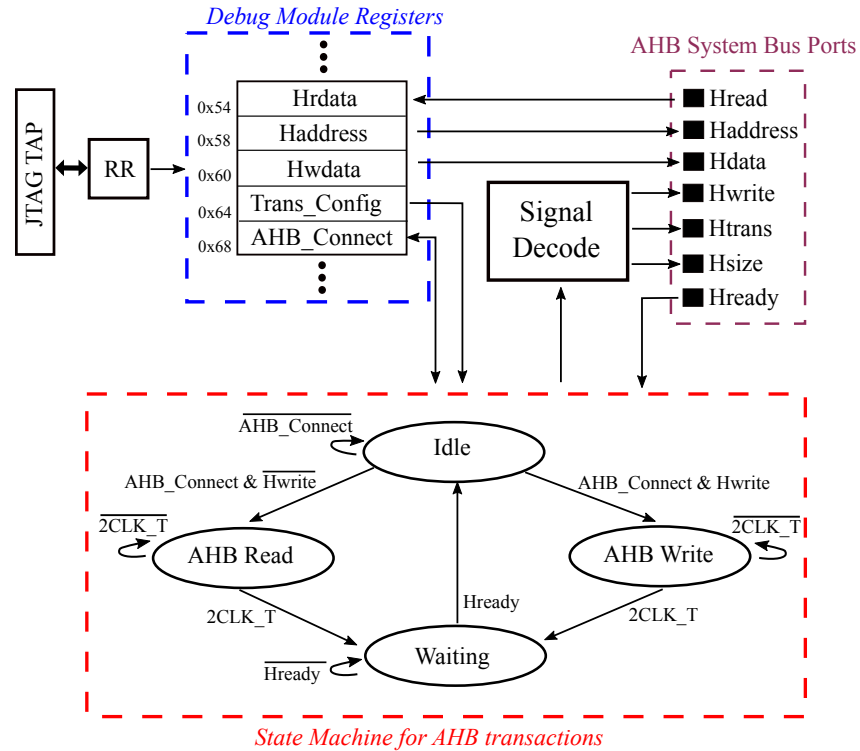


core interruption process to make the pc jump to this memory address. Thereby, the DM makes the core run instructions stored in the program buffer cells. The interruption that performs the mentioned process is caused by the value stored in the *Prog\_Mode* register of the DM. The *Core\_Control* register is responsible for the halt and reset operations. When the halt order is executed, the DM sends a halt signal to the core stalling logic to immediately disable the latch that updates the changes in the pc. Thus, the program counter remains at its value until the halt signal is removed. For the reset order, the DM sends a reset signal to the core reset logic to apply a reset to all circuits of the core. As a final comment, the *Core\_Control* register is described to operate cores with multiple harts as mentioned in section 3.11.2 of (24). The goal of this register is to allow specific harts selections of a core to execute halt, restart and resume operations individually and in groups. In this project, only the options to operate a hart are implemented due to the core implemented in the microcontroller has a single hart.

The designed DM supports bus transactions based on the AHB protocol (25). Figure 15 depicts how the AHB interface of the DM operates. Figure 15 shows the debug module registers which are associated with AHB transactions. Table 3 describes the signals of the AHB protocol which are relevant in this work. The debug module register named *Trans.Config* contains the information of the signals *Hwrite*, *Htrans*, and *Hsize*. The *AHB\_Connect* register is an enabling register that starts an AHB transferring. In case of a reading transaction, the read value is stored in the *Hrdata* register. This register is a read-only register for the RR, while the registers of the address 0x54 to 0x68 are W/R registers. Figure 15 also shows a state machine that controls the transfers through the bus performed by the DM. The state machine keeps the *idle* state until the *AHB\_Connect* register is one. Then, the next state depends on the *Hwrite* value. In *AHB Read* and *AHB Write* states, the AHB ports capture data in DM registers, and a transfer request by bus is performed. Once the state changes to *Waiting*, the *AHB\_Connect* register is modified to zero, and the *Htrans* signal changes immediately to withdraw the transfer request and avoid performing the transfer more than once. To achieve a complete transaction, at least two clock cycles are required. Finally, to return to the *Idle* state, the *Hready* signal must be one.

DM bus transactions might intervene in the system functionality due to the DM is a master of the system bus. Hence, the DM is used for controlling and configuring of the SoC peripherals, but not as a monitoring system. Instead, sample and analysis processes are performed by the BM which presents a non-intrusive operation in the system. DM registers are also employed to configure the BM. Data stored in these registers are used to access signals to the internal configuration registers of the BM.

Figure 15. Behavior illustration of the AHB Bus interface operated by DM registers.

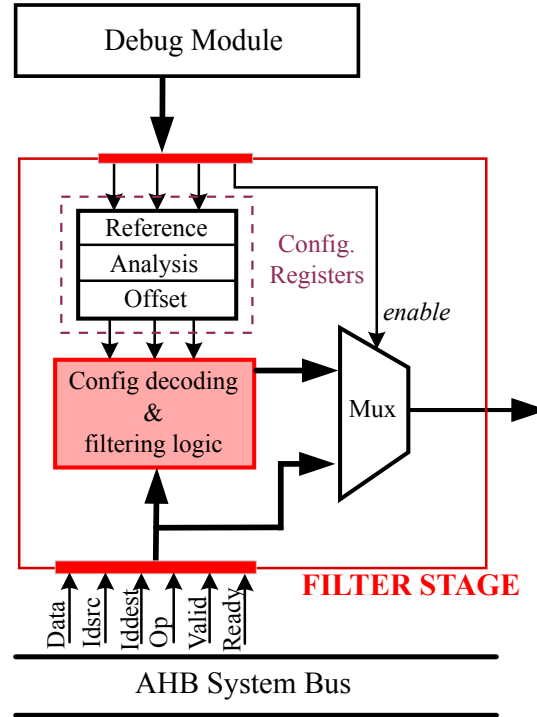


#### 1.4. SYSTEM BUS MONITOR

The System bus monitor is a system that provides non-intrusive monitoring of the bus transactions performed in the SoC. The bus monitor captures bus transactions in order to analyze some features of these. Its architecture is based on a filtering stage and a performance analysis stage. The filtering stage allows performing a detailed or global analysis of the system bus transactions. On the other hand, the performance analysis stage helps to analyze certain characteristics of filtered transactions using only counters.

The filter stage can operate in bypass mode or as a configurable filter. Figure 16 depicts a scheme of the filter stage structure. The bypass mode is the default state which is configured by the control signal of the mux. The filters of the BM

Figure 16. Simplified version of the filter stage in the bus monitor.



are controlled by three configuration registers: *Reference* (32-bit), *Analysis* (10-bit) and *Offset* (32-bit). These registers allow configuring the filters in multiple ways as Table 4 and Table 5 present. The *Reference* register stores a value used as reference in the filtering process. This value is associated with an AHB indicated by the *Analysis* register as Table 5 shows. *Offset* register also contains a reference value used for a second comparison condition. The *analysis* register is divided into two parts. Bits 0-3 indicate the AHB signals (captured from the bus) that are compared to the *reference* and *Offset* registers. For instance, if  $Analysis[3:0]=4'b0$ , then, *Reference* is compared with *data* signal and *Offset* with anyone (indicated with **x**, for this comparison *Offset* does not work). Bits 9-4 indicate the comparison type that the filter performs (Table 4). For instance, if  $Analysis[3:0]=4'b0001$  and  $Analysis[9:4]=6'b000011$ ; then, only transactions with

Table 4. Selection of the comparison type performed by the filter. The comparison is done with *Reference* and *Offset* values.

| Analysis[9:4] | Comparison |  | Analysis[9:4] | Comparison |
|---------------|------------|--|---------------|------------|
| 0             | =          |  | 6             | Range      |
| 1             | >          |  | 7             | Out Range  |
| 2             | ≥          |  | 8             | AND        |
| 3             | <          |  | 9             | OR         |
| 4             | ≤          |  | 10            | XOR        |
| 5             | ≠          |  | Reserved      |            |

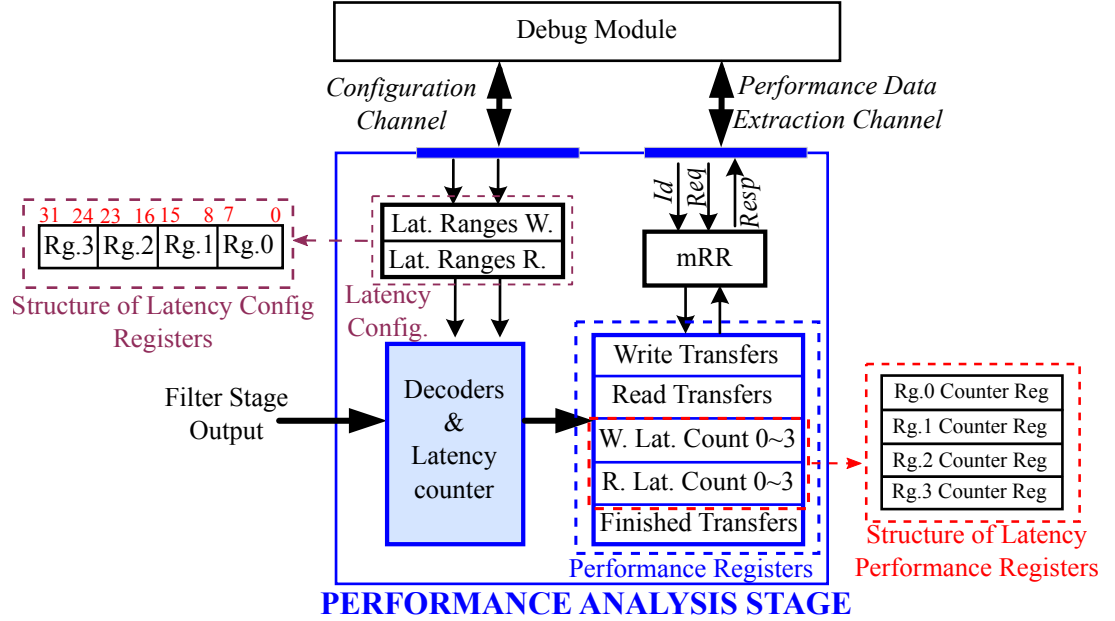
Table 5. Association of *Reference* and *Offset* registers to the AHB signals.

| Analysis[3:0] | Reference      | Offset     |
|---------------|----------------|------------|
| 0 / 5         | Data / Data    | x / Iddest |
| 1 / 6         | ldsrc / ldsrc  | x / Iddest |
| 2 / 7         | lddest / ldsrc | x / Op     |
| 3 / 8         | Op / Iddest    | x / Op     |
| 4             | Data           | ldsrc      |

*ldsrc* > *Reference* pass the filter. In Table 4, comparisons 8, 9 and 10 are only available for the values of *Analysis*[3:0] 4 to 8 in Table 5. For comparisons 6 and 7, the interval is defined as [*Reference*, *Reference* + *Offset*]. These operations are only available for values 0-3 of Table 5.

Performance counters receive transactions selected by the filters as Figure 17 illustrates. Then, they analyze some features of these transactions using counters (18). The performance counters have transfer counters for writing and reading transactions, valid counters, four latency counters for reading operations, and four latency counters for write operations. The latency counters are distributed in ranges. This ranges can be configured in configuration registers of the performance counter. The values stored in the spaces associated with the latency

Figure 17. Simplified version of the performance analysis stage in the BM.

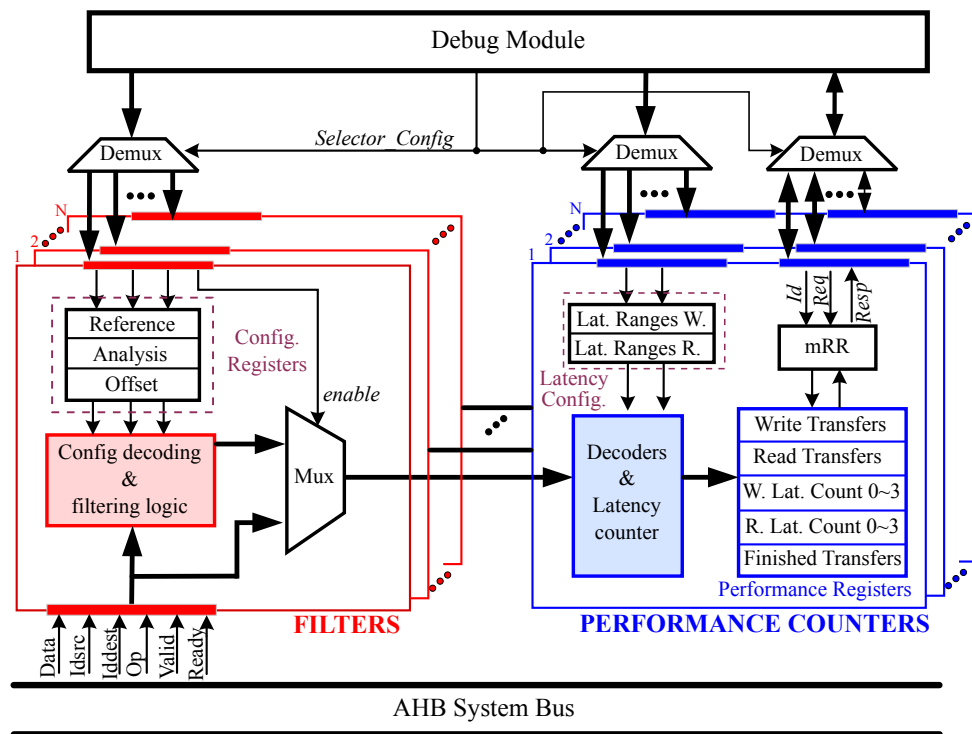


ranges refer to clock cycles. Each latency counter is incremented by one when a transaction has a latency that is within its configured range. To extract performance registers data, a small register router (mRR) is implemented as Figure 18 shows. The DM selects a specific performance counter and sends a read request with an *Id* signal to identify the register to be read.

The design of the system bus monitor is scalable and reusable. If we call an “analysis line” to the combination of a filtering stage and a performance analysis stage, the scalability feature allows synthesizing an SoC with  $n$  analysis lines. Thus, the bus monitor as is depicted in Figure 18, allows performing different analyzes in parallel, each with a different filtering configuration. Some demuxers are added to configure the  $n$  analysis lines of the BM, with the same number of DM registers that would be equivalent to a line of analysis. The reusability feature allows this bus monitor to be reused in future designs and can be easily adapted for another bus protocol.



Figure 18. Top view of the simplified block diagram of the System bus monitor.



## **2. HDL SIMULATION USING THE UNIVERSAL VERIFICATION METHODOLOGY**

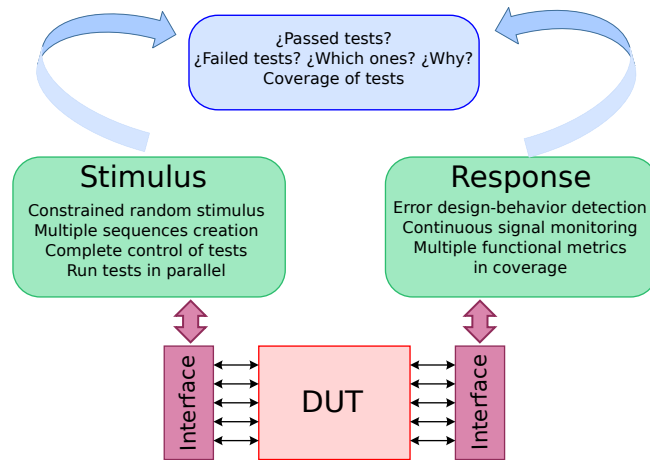
This chapter presents the pre-silicon verification process performed to the debugging platform design presented in the previous chapter. The design is described using Chisel 3.0 language, and this is synthesized to generate a Verilog file of the design. The design described in Verilog is tested using a verification methodology to check all debugging functionalities of the platform. To explain the verification process performed, this chapter begins by exposing the theory of the verification methodology implemented to test the proposed design. Later, the application of the verification methodology for the platform is presented. Finally, simulation results are exposed.

### **2.1. THE UNIVERSAL VERIFICATION METHODOLOGY**

The semiconductor industry developed the Universal Verification Methodology (UVM) as an alternative to offer reliability, reusability, and robustness to functional verification of system-on-a-chip (SoC) design blocks. This methodology quickly became an IEEE standard due to its wide adoption (26). Traditional verification uses static objects from Verilog/SystemVerilog that lack of modularity, restricting code reuse. Another drawback of traditional functional verification is the lack of structured tools to test complex systems with multiple ports and functionalities, causing the simulation to become a tedious and unreliable process. In contrast, UVM applies object-oriented programming properties that allow the creation of dynamic objects to design modular configurable testbenches and verify multiple functionalities either by running automated tests in series or in parallel

(27). Hence, UVM helps to optimize verification time, avoids manual intervention by the designer during test execution, and allows code reuse (7; 28). Figure 19 illustrates the operation logic of UVM for verification IP.

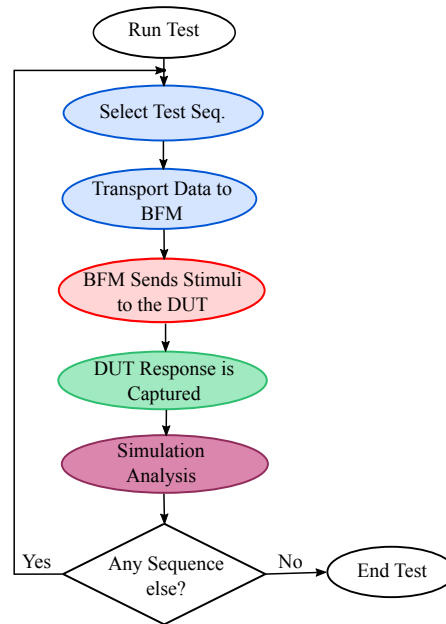
Figure 19. Conceptual illustration of the UVM logic for VIP.



**2.1.1. UVM structure for verification IP** A UVM-based VIP presents the execution sequence shown in Figure 20. A list of test sequences is executed sequentially or in parallel, where stimuli are transported to a bus functional model (BFM) to excite a device under test (DUT). The DUT response is observed and analyzed automatically by analysis modules to ensure the correct behavior of the design. The UVM testbench bases its structure on a hierarchy of modules (one within another) where each module has a different function. UVM offers reusable and maintainable code due to each module performs a single function in the testbench. Then, many parts of the code can be reused for futures designs (29).

A UVM testbench is structured thus: the virtual interface called also Bus Functional Model is a module written using SystemVerilog. This interface is the communication bridge between the DUT and a wrapper of all UVM classes. The BFM

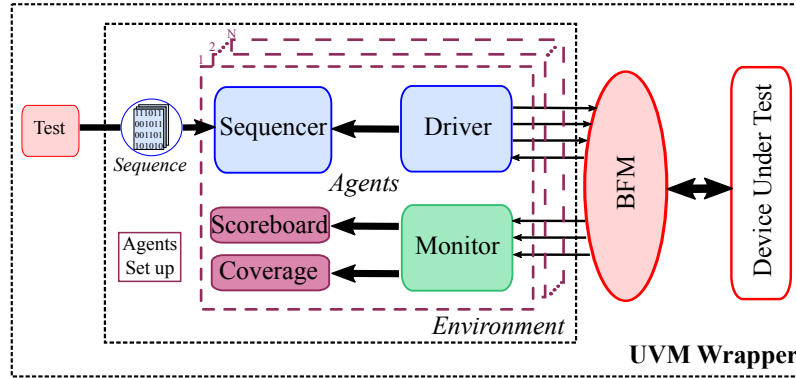
Figure 20. Flowchart of the execution steps of a UVM-based VIP.



is connected to all inputs and outputs of the DUT, controlling the sending/receiving of data to/from this. Then, a BFM handle is passed through components that need to communicate with the DUT. In the other side, the BFM communicates with verification components called agents, responsible for sending and receiving of data in the UVM testbench. Agents contain analysis components as coverers and scoreboards (called self-checkers too), and components that monitor the activity from/to the DUT. In the case of active agents, there are a stimulus maker component and a verification sequences transporter component. They are the sequencer and the driver respectively. Figure 21 shows the agent structure and its sub-components interconnection.

In the next hierarchy level, the environment creates and encapsulates one or multiple agents. This component is also responsible for configuring each one of them as an active or passive agent. The environment passes some configuration objects to the agent, these objects contain information about the agent connec-

Figure 21. Top view of the UVM structure for VIP.



tion. The environment can also have analysis components such as scoreboards. Then, the test encapsulates the environment, configuring the number of virtual sequences to know how many agents will be created. A special function of the test component is to select the stimulus sequences and the order to be sent to the DUT, feeding the sequencer.

Finally, the top-level module instances the DUT, the virtual interface and connects them. A UVM test to be executed is selected. In this module, the packets that contain all the UVM libraries to run the testbench are imported in the simulation configuration.

## 2.2. A UVM-BASED VIP TO TEST THE DEBUGGING PLATFORM HDL DESCRIPTION VIP

To test all functionalities of the debugging system is necessary to design a JTAG interface that converts requests of different instructions to the JTAG protocol (see Table 1). Figure 22 presents a scheme of the UVM-based VIP for the debugging platform. A JTAG interface is designed and adapted as a virtual interface in the UVM-based VIP. This JTAG interface receives specific instructions and data

from the UVM driver. Then, the interface performs the necessary signals changes to load the instructions and data in the respective register of the JTAG TAP. An instruction monitor and a response monitor are responsible for observing the activity to/from the debugging platform. The scoreboard generates a report about the correct/fail transactions, and the Debug Coverage reports how many of the possible cases of stimulation were executed.

Figure 22. Top view of UVM-based VIP for the debugging platform.

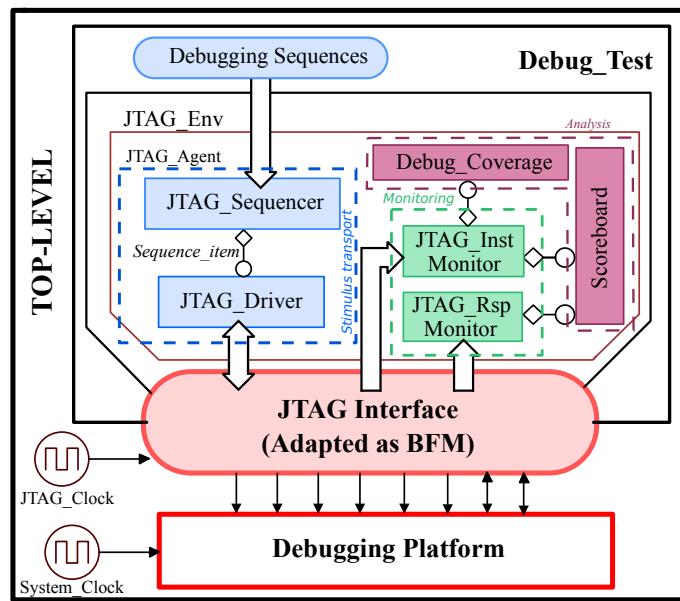
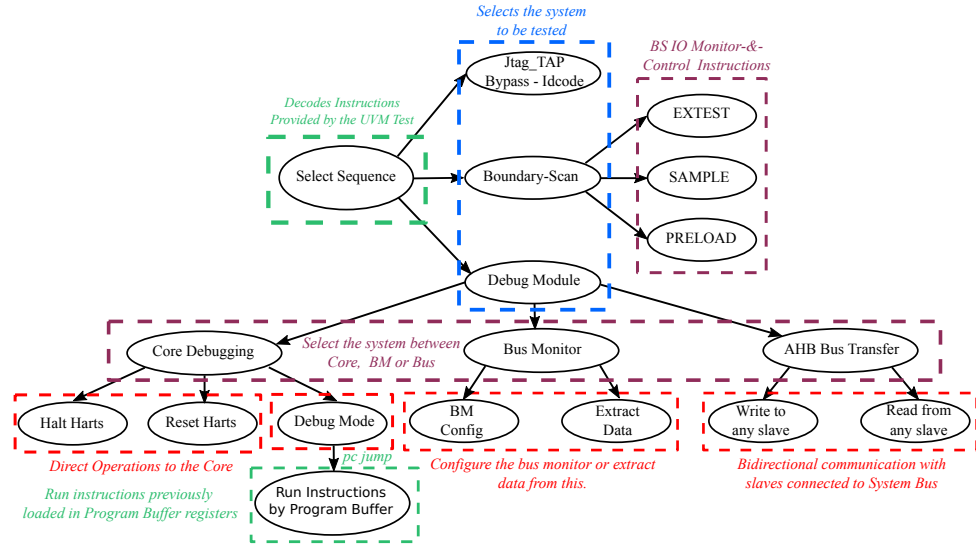


Figure 23 shows all debug module operations. These operations are executed by selecting the DM Access instruction to subsequently write to the data register that corresponds to the debug module data register in the TAP. The address of the DM register to be accessed, the data (for a writing operation case), and the operation (write or read) are loaded in that register. Once this information is loaded, the TAP controller is taken to the *Update\_DR* state, where the debug information is sent to the debug module. For reading operations, data are extracted serially from JTAG TAP. The BFM handles parallelizing this data to perform the checking

Figure 23. Conceptual illustration of the UVM-Based VIP process for the debugging system.



of the behavior of the debugging platform.

Different testing sequences are implemented in the UVM-based VIP to verify all functionalities of the debugging platform. Table 6 presents the implemented sequences. Some constraint rules are added to the configuration of the random sequences to avoid irrelevant data; for instance, data that does not re any instruction or any valid address for the debug module registers. All sequences are parameterized to select the number of times to be executed. Different combinations of the testing sequences can be executed, the sequences are listed and they can be executed sequentially or in parallel.

### 2.3. SIMULATION RESULTS

Synthesis results of the proposed debugging system are presented using two different cases. In both cases, the system is synthesized with four program buffer cells in the DM, and three combinations of filters and performance counters in the BM. Figures 24-to-26 show the results of a run-control case with the option of

Table 6. Sequences implemented in the UVM-based VIP for the debugging platform.

| Sequence     | Description                                                                                        |
|--------------|----------------------------------------------------------------------------------------------------|
| Random       | Whatever operation can be selected                                                                 |
| BoundaryScan | Selects randomly a Boundary-Scan instruction                                                       |
| BS_Exttest   | Runs only the Exttest instruction                                                                  |
| BS_Sample    | Runs only the Sample instruction                                                                   |
| BS_Preload   | Runs only the Preload instruction                                                                  |
| Debug_Module | Selects randomly a debug module operation                                                          |
| DM_WControl  | Writes a random data in the DM direct operation register                                           |
| DM_RControl  | Reads a specific DM register                                                                       |
| DM_JustHalt  | Runs an instruction to halt the core                                                               |
| DM_JustReset | Runs an instruction to reset the core                                                              |
| DM_ProgBuff  | Writes instructions in the Program Buffer                                                          |
| DM_Mode      | Active the debug mode in the core                                                                  |
| DM_AHBBus    | Performs a random write transaction by AHB-Bus followed by a read transaction in the same address. |

running injected instructions directly to the program buffer. Figure. 24 shows the process of halting the core writing to the address  $0x14$ . The register associated to that address controls all debugging operations execution. Figure. 25 describes the process of loading an instruction to the program buffer. Figure. 26 illustrates how the program buffer can be fully loaded to run these instructions by activating a control bit of the program mode. The data synchronization between the JTAG and the DM shown in the previous simulations corresponds to a JTAG clock frequency of 10MHz and a system clock frequency of 48MHz. The synchronization delay grows with the difference of clock frequencies.

AHB system bus transactions are presented in Figures 27 to 28. Figure 27 shows how the DM registers for AHB are loaded. Each register is loaded with a different transaction from the JTAG. The first JTAG-DM transaction loads the slave address in the DM register  $0x58$ . After, the data to be sent is loaded in the



Figure 24. Simulated waveforms of halt core operation of the debug module.

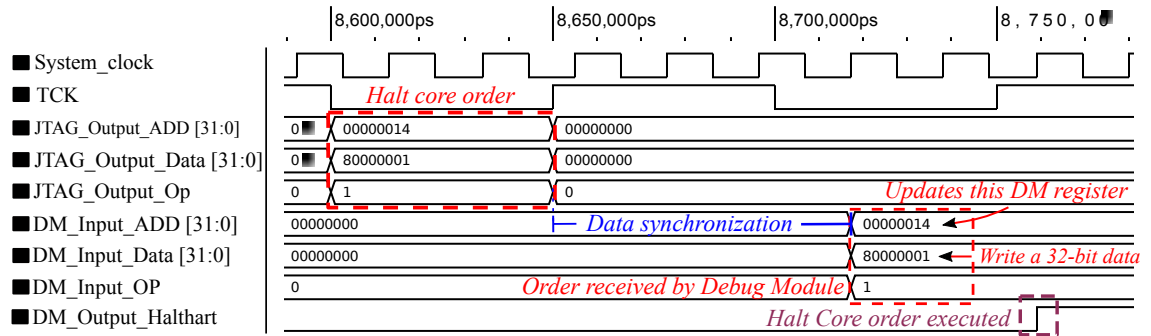


Figure 25. Simulated waveforms of a single loading instruction to a program buffer cell.

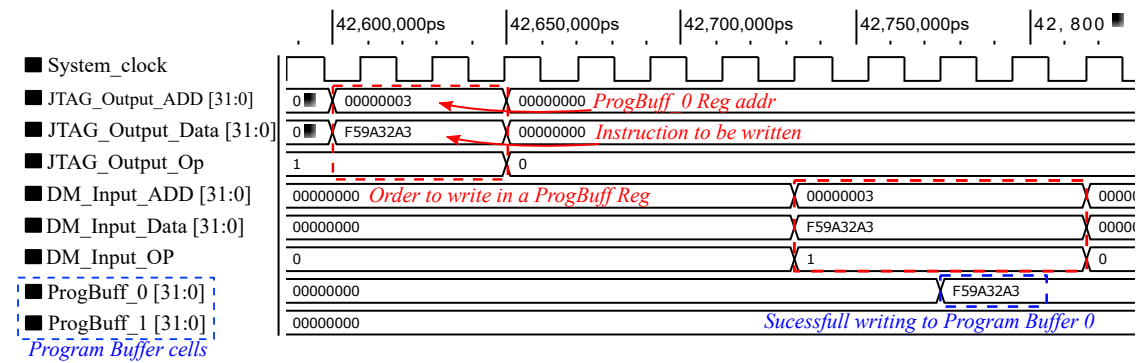


Figure 26. Simulated waveforms of storing instructions in program buffer cells and running these.

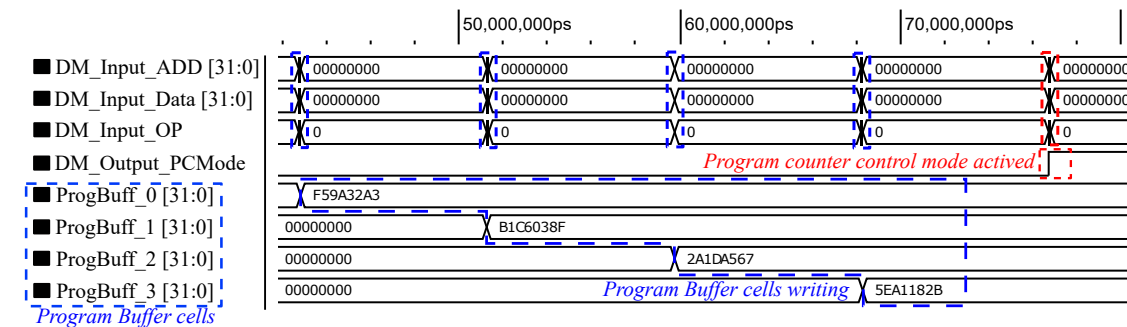


Figure 27. Simulated waveforms of AHB transactions using the debug module.

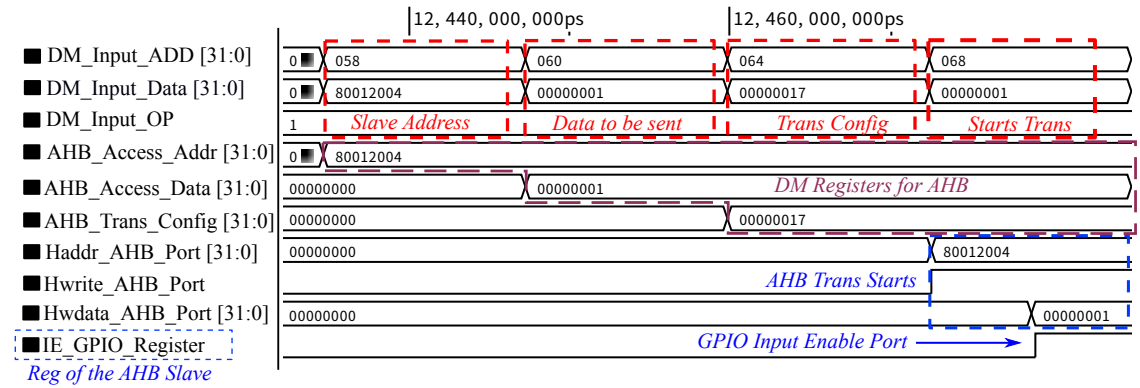
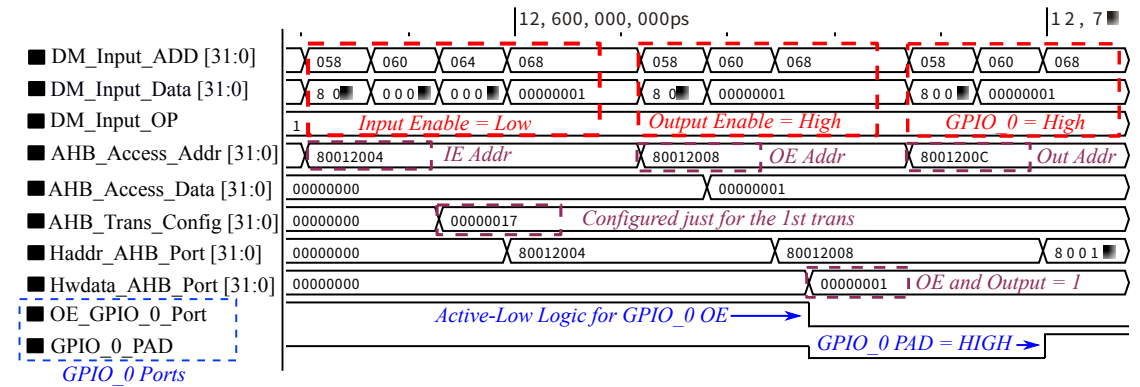


Figure 28. Simulated waveforms of putting GPIO\_0 high by using AHB transfers.



DM register 0x60. The DM register 0x64 contains the data of *HSIZE* and *HWRITE* (1). Finally, the DM register 0x68 enables the AHB transaction. The *IE\_GPIO\_Register* signal in Figure 27 shows how the transaction is successful. Then, the GPIO input enable (IE) port is put on high. Figure 28 presents an example of how to put on high a GPIO pad. Similar processes as in Figure 27 are performed. In this case, three GPIO registers are accessed to operate the GPIO pad. The three transactions presented in Figure 28 shows *IE = Low*, *OE = High*, and finally, *GPIO\_Out = High*. The figure shows how the GPIO pad is successfully put on high.

Figure 29. Simulated waveforms of an AHB reading transaction.

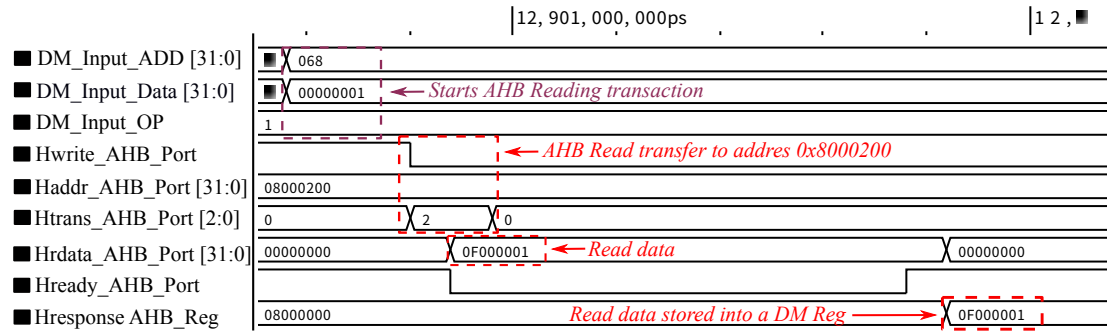
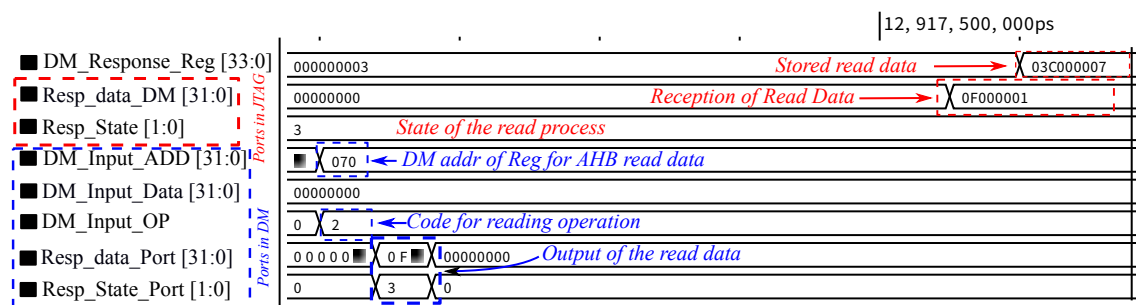


Figure 30. Simulated waveforms of reading transaction to the DM register for AHB response.



The process for AHB Reading transactions is presented in Figures 29 and 30. Figure 29 shows an AHB transaction putting *Hwrite\_AHB\_Port* signal on low to indicate a read transaction to the address *0x8000200*. The data to be read is received in the *Hrdata\_AHB\_Port*. When *Hready\_AHB\_Port* returns to high, the read data is stored in a DM register reserved for AHB responses. Figure 30 illustrates the process to extract the data out of the chip. A reading operation is performed to the DM to obtain the data stored in the DM register for AHB responses. The data is returned to the JTAG by the *Resp\_data\_DM* port. Additionally, a *Resp\_state* signal is returned to indicate if the operation was successful. These two signals are concatenated and stored in a data register. Finally, the data stored in this register are serially extracted out of the chip in a *Shift\_DR* state.

A detection filter case using bus monitors is described in Figures 31 and 32. Figure 31 shows a monitoring operation using the filtering configuration described. In this case, a filtering is applied to the destination addresses, where only addresses 100 to 150 (decimal) pass the filter stage. The transactions that pass the filters are analyzed by performance counters associated with transactions and latency counts. The timing diagram illustrates how only transactions that satisfy the filter settings increase the transaction counter. Also, depending on the number of clock cycles taken by these transactions, one of the latency counters increases. A more detailed analysis is shown in Figure 32 where filtering configuration is set to a specific destination address and a specific data.

The testing sequences applied to the debugging platform, the number of repetitions of each sequence and their results are presented in Table 7. Some sequences were combined to be executed one after the other and to repeat the procedure a certain number of times. As in the case of executing the *Preload* instruction to subsequently execute *Extest*. The results column shows the general

Figure 31. Simulated waveforms of monitor operation with filter config:  
Comparison:IDsrc, Operation: range, Reference: 100 offset: 50, Latency range  
W0:  $\leq 5$ , Latency range W1:  $[6, 10]$ .

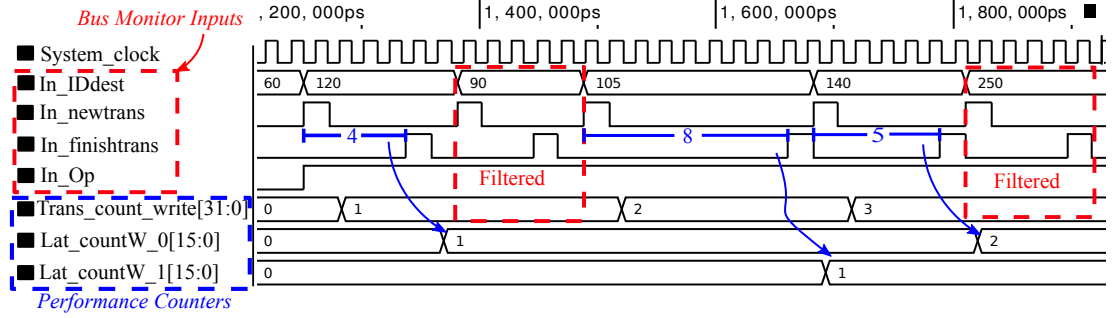
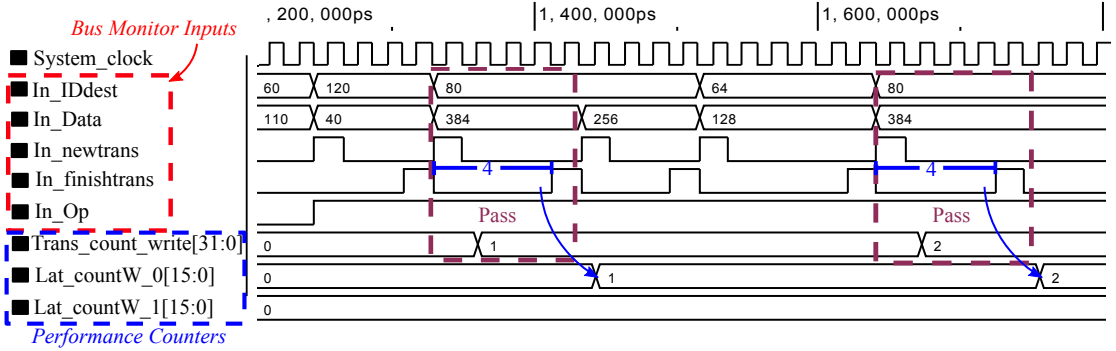


Figure 32. Simulated waveforms of Monitor operation with filter config:  
Comparison:IDsrc, Operation: AND, Reference: 80 (slave id) Offset: 384(Data).  
Latency range W0:  $\leq 5$ , Latency range W1:  $[6, 10]$ .



decision of the testbench based on all the repetitions of a given sequence. The presented debugging system passed all the tests performed by the VIP system. The sequences present different numbers of repetition, these numbers were chosen taking into account according to the test and the amount of possible stimuli that could be applied. The selected values and executed sequences allowed to obtain the coverage analysis results that are presented in Table 7. These parameters shown in Table 7 were chosen due to they are the tests necessary to ensure that the design operates correctly in all its functions.

Table 7. UVM results of debugging platform verification.

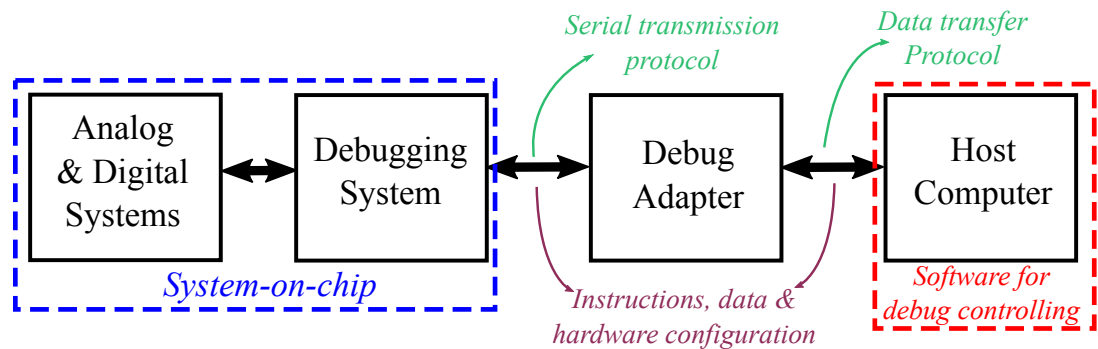
| <b>Sequence</b>              | <b>Repetitions</b> | <b>Result</b> |
|------------------------------|--------------------|---------------|
| Random                       | 2000               | Pass          |
| BoundaryScan                 | 1000               | Pass          |
| BS Preload<br>& BS Extest    | 100                | Pass          |
| BS Sample                    | 100                | Pass          |
| Debug Module                 | 1000               | Pass          |
| DM_WControl<br>& DM_RControl | 1000               | Pass          |
| DM_JustHalt                  | 10                 | Pass          |
| DM_JustReset                 | 10                 | Pass          |
| DM_ProgBuff<br>& DM_Mode     | 1000               | Pass          |
| DM_JustHalt                  | 1000               | Pass          |

### 3. SOFTWARE INTERFACE FOR DEBUGGING

Up to this point, this thesis has described the structure of the proposed debugging system, and the functional verification process performed to test the hardware design. Now, this chapter shows the research work done at the software level. Here, a software interface is described to operate the debugging system hardware. First, the chapter exposes some methods currently implemented in the industry to operate software-level debugging systems. Later, a software interface is described to operate the proposed debugging platform. Hardware connections, devices and implemented libraries for the software interface are presented. Additionally, the chapter exposes some code examples using the implemented libraries.

#### 3.1. SOFTWARE FOR DEBUGGING

Figure 33. Simplified diagram block of the hardware-software communication.



On-chip implemented debugging systems require software that configures and operates debug tasks such as run-control debug, memory access, internal system access, among others (30). Hence, the instructions, data and the operation

mode of a hardware designed for debugging purposes, are controlled through a software interface as shown in Figure 33. The interface must send instructions described using a programming language or a machine language to the debugging system. These instructions are commonly sent to the SoC by using external hardware as debug adapter. This adapter converts the instructions described in a programming language to data transfer sequences of a serial protocol supported by the SoC (JTAG for example).

The interaction between software and hardware in SoCs implies the knowledge of both parts. Developing, debugging and improving software running on an SoC requires access to the SoC internal structures (31). In other words, describing the algorithms that allow operating a debugger requires to consider architectures, binary codes, execution steps, internal processes and signals of the SoC debugging hardware. Moreover, having knowledge of the processor and buses implementation of the SoC allows describing control and access algorithms for the SoC peripherals.

This section presents some concepts and techniques implemented in the industry at a software-level for SoC debugging. First, a common tool adopted in different debugging techniques called the GNU debugger is presented. GNU debugger servers are commonly used to debug integrated systems at the software level. Later, these servers can be observed in two examples to be presented in this section: Open SoC Debug that proposes a structure for debugging, and the Open On-Chip Debugger which is one of the most common tools used for devices debugging.

**3.1.1. The GNU debugger** The GNU debugger(GDB) is free software that allows monitoring the activity of another program while it is executed. GDB allows



observing the execution of a program in steps configured by the user. Moreover, when a program crashes, GDB helps to visualize the activity of the program (32).

These are the four main kinds of operations performed by GDB:

- Start a program, specifying some parameter that might affect the program behavior.
- Stop a program on specific conditions.
- Examine the processes performed by a program when it is stopped.
- Modify the program in order to test with the effects of correcting a bug. Then, the program can continue with other bugs.

The programs that are executed in the debugging process might be running on the same machine as GDB (native), another machine (remote), or a simulator. GDB can run on most popular UNIX and Microsoft Windows variants, as well as on Mac OS X. The programs can be written in some of the next languages supported by GDB (33):

- |            |               |
|------------|---------------|
| • Ada      | • Objective-C |
| • Assembly | • OpenCL      |
| • C/C++    | • Modula-2    |
| • D        | • Pascal      |
| • Fortran  | • Rust        |
| • Go       |               |

GDB servers are commonly used to debug integrated systems at the software level. This is exposed in the next subsections that two examples of software implementation for hardware debugging.

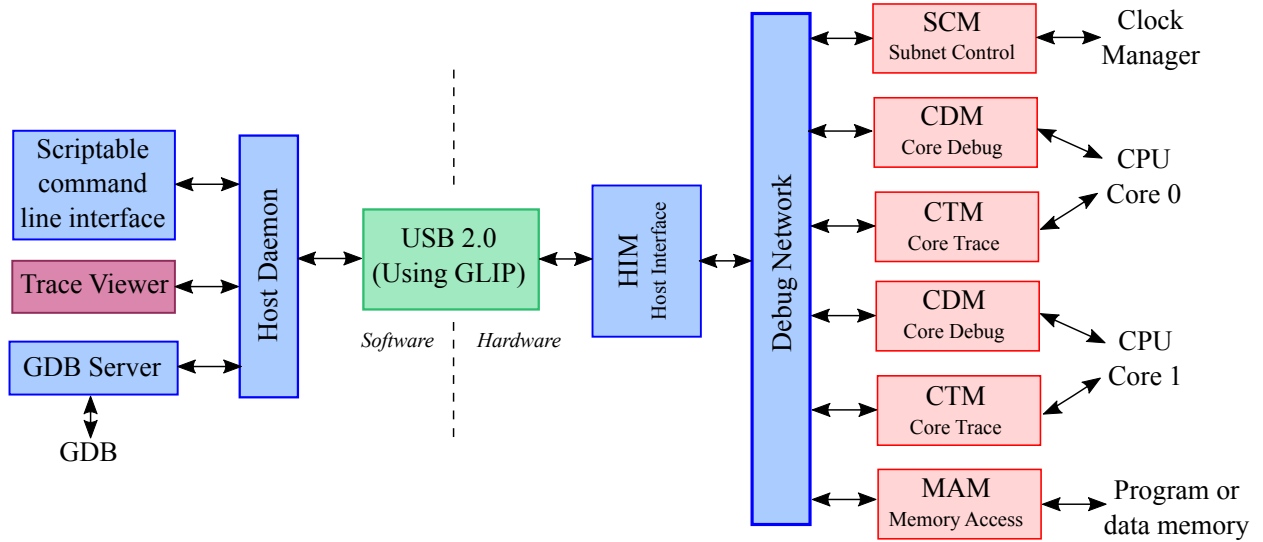
**3.1.2. Open SoC Debug** Open SoC Debug (OSD) is a project co-managed by the OpTiMSoC and lowRISC projects where the main goal is to establish a repository of common SoC debug components (34; 35; 36). Among other goals, OSD seeks to create a specification of a unified infrastructure for SoC run-control and trace debugging, and to provide a generic hardware for debugging. To achieve the goals, the specification defines an extensible debug system architecture

Figure 34 presents an example of OSD structure for debugging a dual-core system (31). Figure 34 shows the division between hardware and software where the connection between both levels can be either JTAG or USB using GLIP (Generic Logic Interfacing Project) (37). In a software-level, OSD shows a host daemon which encapsulates the communication to the device and provides an API for various tools communicating with the debug system. To control the system, OSD implements a scriptable command-line interface (SCLI), where different operations (such as reset, halt, etc) can be executed. Also, this SCLI can be used to read and write memories. A GDB server provides an interface between the GDB and the central debugging functionality. Then, the software can be debugged with unmodified GDB (and other GDB-enabled integrated development environment).

**3.1.3. Open On-Chip Debugger** The open on-chip debugger (OpenOCD) is one of the most common tools used for devices debugging. OpenOCD was created as part of a diploma thesis at the University of Applied Sciences Augsburg (38), and seeks to offer a free and open On-chip debugging in-System programming and boundary-Scan testing.

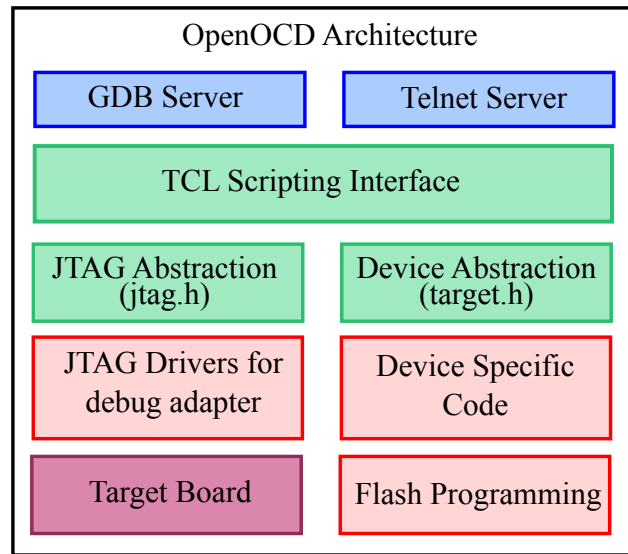
OpenOCD architecture is designed to provide easy interaction with the device under debug and the hardware used in the connection. Figure 35 displays the OpenOCD architecture. Two servers are presented inside OpenOCD: A GDB

Figure 34. Example of debugging of a dual-core system using OSD.



server and a telnet server. The GDB server allows connecting and interacting with the GNU debugger. On the other hand, the telnet server allows users to connect to the server process using a telnet client. Thereby, different users can operate a single debug system at possibly remote locations. A TCL scripting interface is provided by OpenOCD to digit TCL commands and execute different operations for debugging. To debug a specific device (for example an FPGA or a microcontroller), some information is required to communicate with this and operate its internal hardware. Accordingly, a device abstraction is provided and described in libraries and header files (as target.h). As presented in Chapter 2, JTAG has a detailed protocol and a specific binary code to manage the different instructions supported in the design. Hence, a JTAG abstraction is also required in libraries and a header file (jtag.h). Additional hardware is required to connect with the hardware to be debugged. Debug adapters are responsible for receiving orders from the host and for generating the signal sequence of the JTAG proto-

Figure 35. Top-level architecture of OpenOCD.



col to comply with said orders. Communicating and operating this debug adapter requires additional drivers. A device-specific code allows working with all functions of the device (halt/resume step, etc), and the flash programming is used for memory management.

Currently, OpenOCD supports debugging for different families of devices as microcontrollers and FPGAs, and contains drivers to connect with different debug adapters found in the market (FT2232, ZY1000, etc) (39). Figure 36 shows a connection diagram of a debugging process, where hardware and software sides are indicated. In the software side (running in the host computer), GDB and OpenOCD interact by using the GDB remote serial protocol (RSP). RSP is a general purpose protocol for connecting to remote targets based on a simple text command-response protocol. OpenOCD can transmit orders to the debug adapter by using USB, Ethernet or a parallel data transmission protocol. The debug adapter contains a communication interface to decode these orders and generate signaling of the protocol to be used for connecting the development

Figure 36. Example of the debug connection using OpenOCD (simplified version).

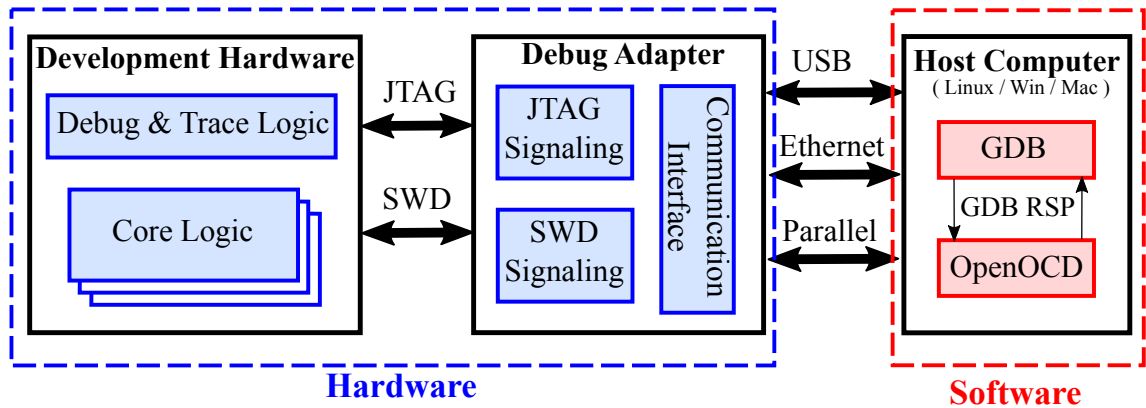
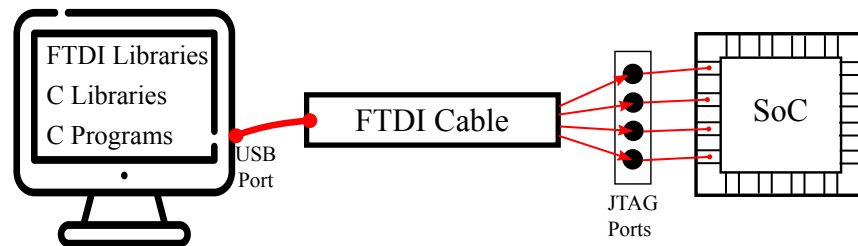


Figure 37. Illustration of software-hardware connection.

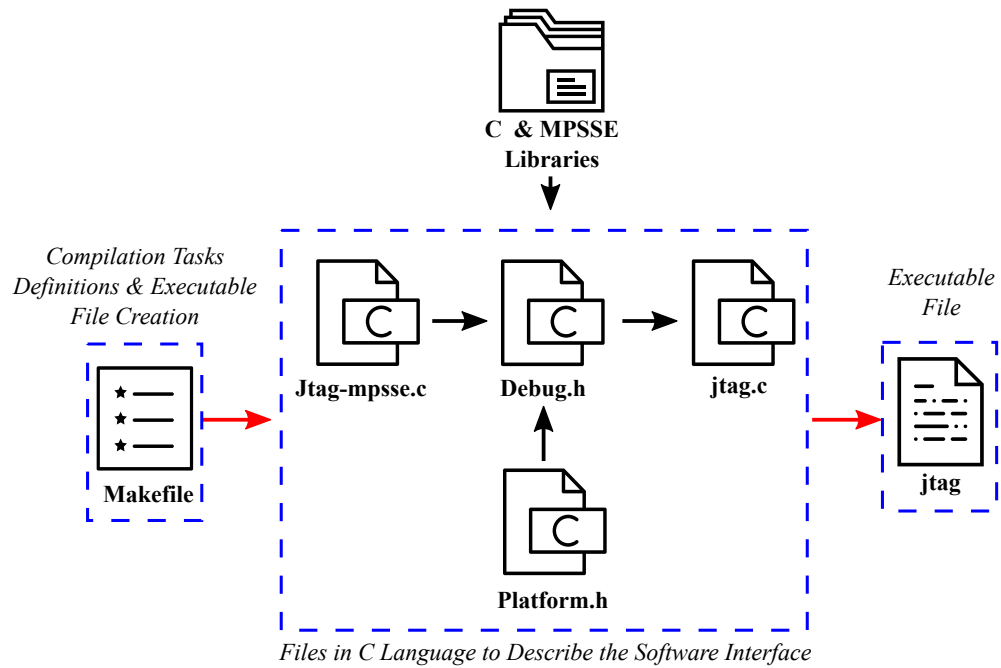


hardware. This protocol can be JTAG or serial wire debug (SWD) (40).

### 3.2. IMPLEMENTED SOFTWARE INTERFACE

Controlling the debugging platform exposed in Chapter 2 requires a description of the JTAG signaling. The different operation types performed by the platform involve a specific behavior in the signals of the JTAG protocol. The processes shown in Chapter 3 to stimulate the debugging system in simulation, must now be developed in software. Different functions must be implemented in order to cover all the actions of the debugging system. These functions must be integrated into a library in order to describe programs easily by calling the library functions. This section has the purpose of serving as a guide to use this software interface.

Figure 38. Conceptual diagram of the composition of files set-up for the software interface.



Hence, the section presents a description of the software interface developed by showing how it is configured and how it can be used to operate the platform.

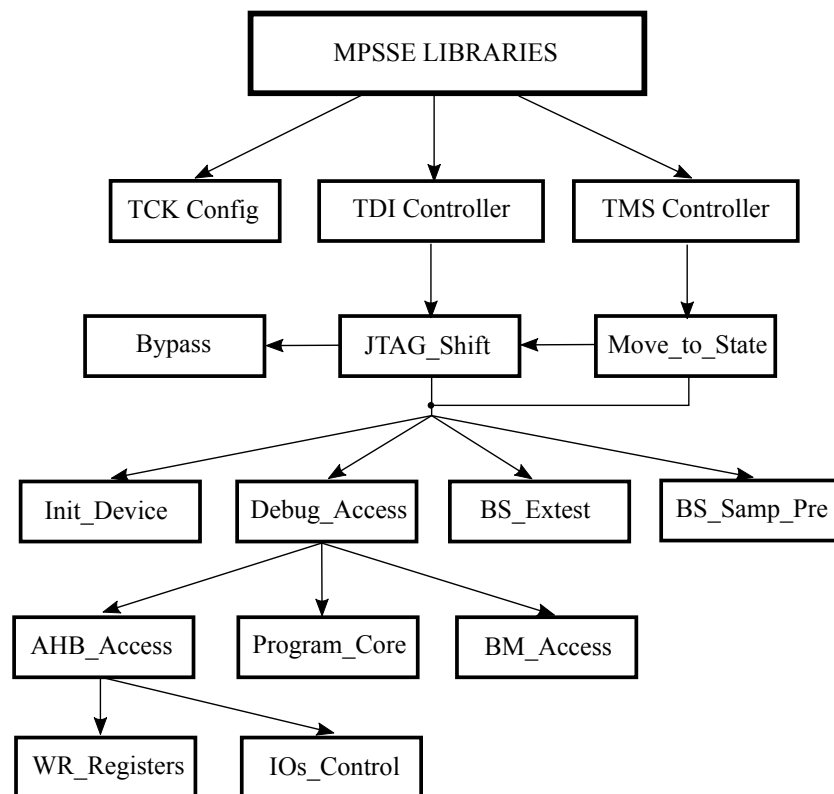
Figure 37 illustrates a scheme of the implemented software-hardware connection. In this work, a USB Hi-Speed-MPSSE cable is used as debug adapter to send commands to the JTAG ports of the SoC. This cable contains an FT232H single-channel hi-speed USB to multipurpose UART/FIFO IC. MPSSE libraries are required in the computer where different programs are described to send data to the SoC (41). These libraries allow programming the behavior of the JTAG protocol signals. Thus, the software implementation work performed in this thesis consists of describing all library functions to control the actions of the debugging platform.

Descriptions and creation of the software interface are made by different files as Figure 38 depicts. C and MPSSE libraries are required in the files described

in C language, which conform the software interface. The file *Jtag-mpsse.c* contains all functions programmed to perform operations in the designed debugging platform. The file *Platform.h* contains information about the debugging platform configuration. This information includes: binary codes of JTAG instructions, addresses of control-and-data registers of the microcontroller peripherals and circuits, addresses of the debug module registers, and binary codes for debug module tasks. The files *Jtag-mpsse.c* and *Platform.h* are included in the header file *Debug.h*. The file *Debug.h* declares all functions that are used to create a program for the debugging platform. Then, this file is included as a library in the main program to allow executing all debugging functions. The file *jtag.c* contains the function *main()* where the debugging instructions are programmed. The *Makefile* compiles all described files and generates an executable file *jtag*. The executable file contains the machine code to be sent to the debug adapter.

The construction of the functions described in *jtag-mpsse.c* is presented in the diagram of Figure 39. MPSSE libraries allow creating functions to control TDI and TMS signals in order to implement the sending of data by TDI and control the transitions of the JTAG TAP controller. Additionally, the TCK clock signal frequency can be configured. *JTAG\_Shift* and *Move\_to\_State* allow creating all functions necessary to operate the different functions of the debugging platform. *Init\_Device* generates a sequence of recognition of devices connected by USB to the PC. An ID code is obtained applying a bypass sequence to the device. *BS\_Exttest* and *BS\_Samp\_Pre* allow performing boundary-scan operations and *Debug\_Access* allows performing writing and reading transactions to the debug module registers. Using the *Debug\_Access* function, *AHB\_Access*, *Program\_Core* and *BM\_Access* are created. *AHB\_Access* performs writing and reading transactions by the system bus, allowing to access to any register connected to

Figure 39. Composition of C libraries and functions to control the software interface.





the bus. Also, a function to implement different applications using the IOs of the chip is implemented. *Program\_Core* performs debug module writings in order to load instructions in the program buffer to be subsequently executed by the core. Finally, *BM\_Access* indicates some functions that allow configuring the bus monitor or extract data from it.

Figure 40. Representation of the C functions to operate the debugging platform.

|                                                                                                                                                                                                                                        |                                                                                                                                                                                                                                   |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>void Debug_write (JTAG *jtag, u32 data, u32 address, Bool wr){     /* *jtag : jtag_handle        data: Data to be written in DM Registers        address: Address of a DM Register        wr: Write(true), Read(false) */ }</pre> | <pre>u64 SAMPLE_PRELOAD (JTAG *jtag, u64 data, int space){     /* *jtag : jtag_handle        data: Data to be sent serially through the BS Register        space: Number of bits to be sent */ }</pre>                            |
| <pre>u32 EXTTEST_INST(JTAG *jtag, u32 data, int bits){     /* *jtag : jtag_handle        data: Data to be sent serially through the BS Register        space: Number of bits to be sent */ }</pre>                                     | <pre>void AHB_Access (JTAG *jtag, u32 address, u32 data, Bool wr){     /* *jtag : jtag_handle        data: Data to be sent by system bus        address: Address of the peripheral        wr: Write(true), Read(false) */ }</pre> |
| <pre>void GPIO_REG(JTAG *jtag, u32 offset, u32 value){     /* *jtag : jtag_handle        offset: Value to select a specific register of the GPIO        value: value to be written in the specific GPIO register */ }</pre>            | <pre>void GPIO_WRITE(JTAG *jtag, u32 value){     /* *jtag : jtag_handle        value: value to be written in the GPIOs */ }</pre>                                                                                                 |
| <pre>void BM_Config_Filter (JTAG *jtag, u32 Reference, u32 Analysis, u32 Offset, int LA_num){     /* *jtag : jtag_handle        Reference, Analysis, Offset: Filter Registers        LA_num: Analysis line to be configured */ }</pre> | <pre>void Program_Core (JTAG *jtag, u32 Inst, Bool En){     /* *jtag : jtag_handle        Inst: 32-bit Instruction        En: Active PC_Mode (true) */ }</pre>                                                                    |
| <pre>void BM_Extract_PC (JTAG *jtag, int Register, int Filter_num){     /* *jtag : jtag_handle        Register: Performance Register to be read        LA_num: Analysis line to be read */ }</pre>                                     | <pre>void BM_Config_PC_Latency (JTAG *jtag, u32 Lw, u32 Lr, int LA_num){     /* *jtag : jtag_handle        Lw, Lr: Write and Read latency respectively        LA_num: Analysis line to be configured */ }</pre>                   |

Figure 40 presents the functions implemented in C language to operate the different functionalities of the debugging platform. Each function of Figure 40 indicates the input parameters and what means each parameter. These functions conform the library of the debugging platform. Figure 41(a) describes a program-

ming example using the mentioned library which is included in the header file *debug.h*. The programming example is known as "blinking" which consists of putting in high and low a GPIO repeatedly to turn on and off a LED. Another example is presented in Figure 41(b), this shows a simple program to configure the system bus monitor, in this case, the program sets the next configuration for the filter of the Analysis line 0:  $Analysis[3 : 0] = 4$  and  $Analysis[9 : 4] = 6$  which indicates that only transactions in the range [0x64, 0x66], indicated by [Reference, Reference + Offset], pass the filter. Also, the latency is configured for write and read transactions: Latency Range 0: [1, 5], Latency Range 1: [6, 10].

Figure 41. Code examples of programming the debugging platform.

```
#include <stdio.h>
#include "debug.h" //debugging library
int main(int argc, char **argv) {
    JTAG *jtag; //Create a jtag handle

    if (!(jtag = jtag_open())) //Detect and open device
        return -1;
    int a = 1;
    u32 b;
    while(1){
        for(int i = 0; i < 1000; i++){
            a = !a;
            b = (a) ? 0xFFFF:0x0; //Turn on/off all GPIOs
            GPIO_WRITE(jtag, b); //Function of the library
        }
    }
}
```

(a) Blinking

```
#include <stdio.h>
#include "debug.h" //debugging library
int main(int argc, char **argv) {
    JTAG *jtag; //Create a jtag handle

    if (!(jtag = jtag_open())) //Detect and open device
        return -1;

    BM_Config_Filter ( jtag, 0x64, 0x2, 0x0, 0);
    BM_Config_PC_Latency (jtag, 0xA05, 0xA05, 0);
}
```

(b) BM Configuration

This chapter has exposed a software interface to control the proposed debugging platform. The functions developed for this interface are compiled to check the desired behavior. Nevertheless, ensuring that the software interface can operate the debugging functions of the platform is only possible in a physical implementation. Accordingly, the following chapter presents implementation results, where the software interface is applied to control a physical implementation of the de-

bugging system.

## 4. IMPLEMENTATION RESULTS

This chapter exposes the implementation results of the proposed debugging platform. The chapter begins with the synthesis results of the proposed design. To perform the experimental tests, the platform is integrated within a 32-bit microcontroller. Section 5.2 presents experimental results performed to a silicon implementation of a microcontroller that includes part of the debugging platform. The section describes the set-up and test plan, and presents a discussion of the measurement results. The experimental results show a fail in the JTAG that avoids the execution of debugging actions. Then, the platform is redesigned in order to solve the measurement issues. In Section 5.3, an enhanced version for the complete debugging system within the 32-bit microcontroller is implemented in an FPGA. The software interface presented in Chapter 4 operates the FPGA through implemented JTAG ports that are assigned to FPGA GPIOs.

### 4.1. SYNTHESIS RESULTS

The debugging platform is synthesized within a 32-bit microcontroller of 1.6mm x 1.6mm as shown in Figure 42. Figure 42 displays the pads associated with the design and the area occupied by the debugging system. The platform is synthesized with four program buffer cells in the DM, and three combinations of filters and performance counters in the bus monitor. The design synthesis obtained an area of  $148,049\mu m^2$  using 4401 gates equivalent in  $0.18\mu m$  CMOS as Table 8 shown. The debugging system occupies 5.8% of the total area of the microcontroller.

The debugging platform is designed with properties that allow debugging ac-

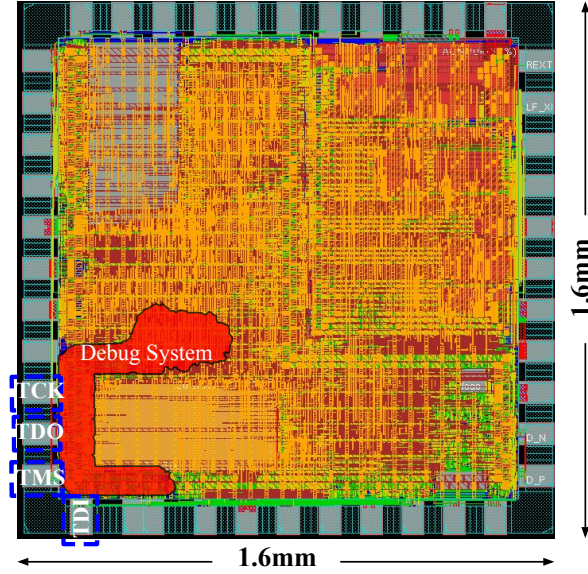
Table 8. Resources Comparison with Other Debugging Systems.

| Feature           | Proposed<br>(0.18 $\mu m$ ) | (14)<br>(0.13 $\mu m$ ) | (16)<br>(FPGA) | (42)<br>(NA) |
|-------------------|-----------------------------|-------------------------|----------------|--------------|
| Gates Eq.         | 4401                        | 7300                    | FPGA           | 17341        |
| Area[ $\mu m^2$ ] | 148,049                     | 36,849                  | FPGA           | NA           |
| % chip            | 5.8%                        | NA                      | >10%           | 33%          |

tions to a microcontroller of low-area. Hence, the dimensions of this debugger are small since its cost in area must be relative to the dimensions of the chip for which it is being designed. In this case, the debugging system occupies 5.8% of the total area (Table 8). This chip contains a core of single hart, two buses, and the peripherals shown in Figure 1.2. Table 8 provides a comparison with other three debugging systems reported in the literature. In this Table, the compared works do not present the same fabrication technology. Thus, we will focus on analyzing the area in gates equivalent and the percentage of area in the SoC. Table 8 shows that the proposed design has a lower number of gates equivalents than systems in (14) and (42), which present SoCs with multi-core features. These two systems require a greater number of control modules (or a larger module) to debug the multiple SoC cores. The implementation of test buses, on-chip networks, and debug cores for multiple systems is a useful solution that tend to increase the area of the debugger. About the overhead area, the work in (16) presents a platform according to the needs of the SoC and reports an overhead area around 10%. Otherwise, the work in (42) reports an area overhead of 33%, due to the platform is done for SoCs of greater dimensions and complexity.

The design of the proposed debugger is scalable, reusable and easy to adapt to new SoCs. The platform is based on a central debugging module that contains a bank of registers that control all processes. Thus, new features can be easily

Figure 42. Debugging platform synthesized into a 32-bit microcontroller.



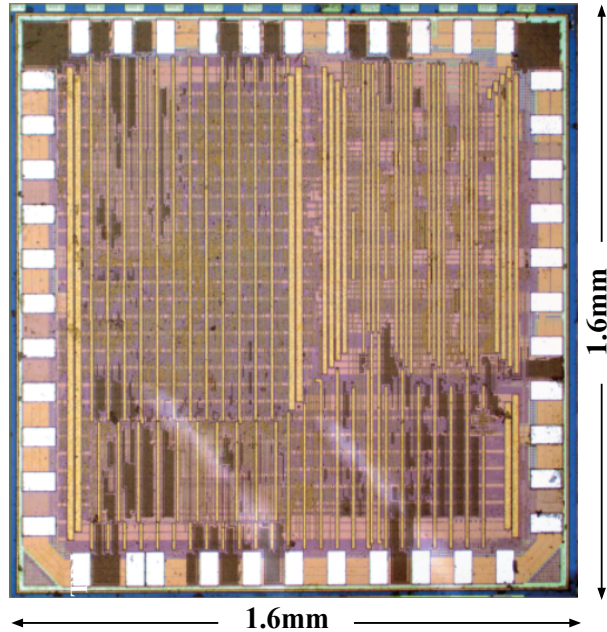
added to the exposed platform. Adding registers and operation interfaces does not imply redesigning the structure of the debugger. In other words, the communication to the new control registers in the DM can always be executed in the same way as presented in this thesis.

As a synthesis results conclusion, the debugging system has dimensions according to the presented microcontroller (5.8%). Future microcontroller designs that add new features and systems can be adapted to this debugger. Moreover, the platform can be easily modified to cover the needs that those new systems bring with them.

## 4.2. EXPERIMENTAL RESULTS

A part of the exposed debugging system was implemented in silicon within the 32-bit RISC-V based microcontroller presented in Figure 5. The JTAG and the DM were implemented in this tapeout. The 32-bit microcontroller is fabricated

Figure 43. Micrograph of the 32-bit microcontroller silicon implementation.



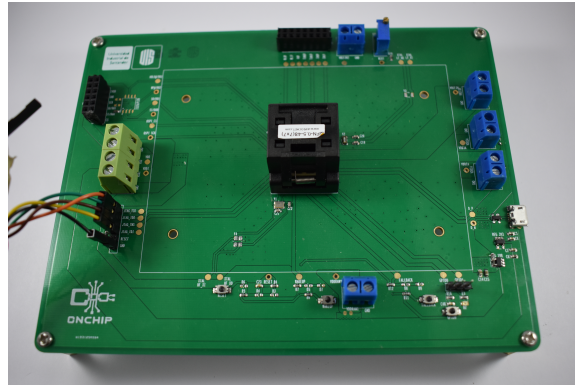
in TSMC  $0.18\mu m$  CMOS technology by Europractice company (43). The company returns dies and encapsulated chips. Figure 43 shows a naked die of the fabricated chip. This microcontroller project is named *Guerinii*. To test the microcontroller, some test boards are developed by the integrated systems research group OnChip at Universidad Industrial de Santander (UIS). Figure 44 shows the test boards built to test the microcontroller. In Figure 44(a), a chip encapsulated in QFN48 is put into a socket, thereby many chips can be checked. The board of Figure 44(b) contains the encapsulated chip ready to be tested. Both boards contain elements necessary to power supply the chip, to establish JTAG and QSPI communication, and to take measurements of the chip operation.

This section exposes the experimental results of applying tests to this version of the platform fabricated in silicon and composed by the JTAG and DM. First, the set-up of the performed test is presented by indicating the used software and hardware, and the connection of them. Later, the section describes the planning

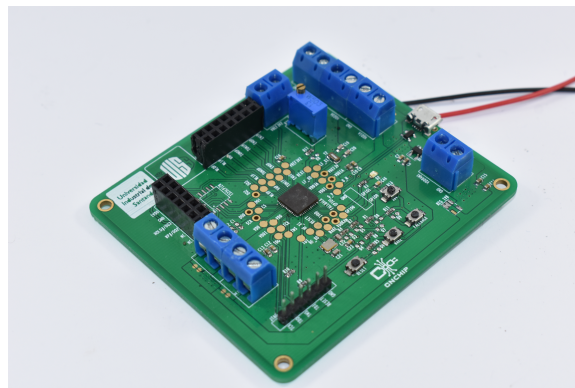


of the tests to be applied, and the features of the taking of measures, besides the measurement results. A fail found in JTAG precludes the realization of all the tests proposed. The section ends with a discussion about the experimental results obtained for this platform version.

Figure 44. Test boards for the 32-bit microcontroller chip.



(a) Test board with socket.



(b) Test board with QFN48 package chip.

**4.2.1. Set-up for experimental measurements** This subsection describes the connections, instruments and configuration of the experimental tests. The equipment required to perform the experimental tests are mentioned below:

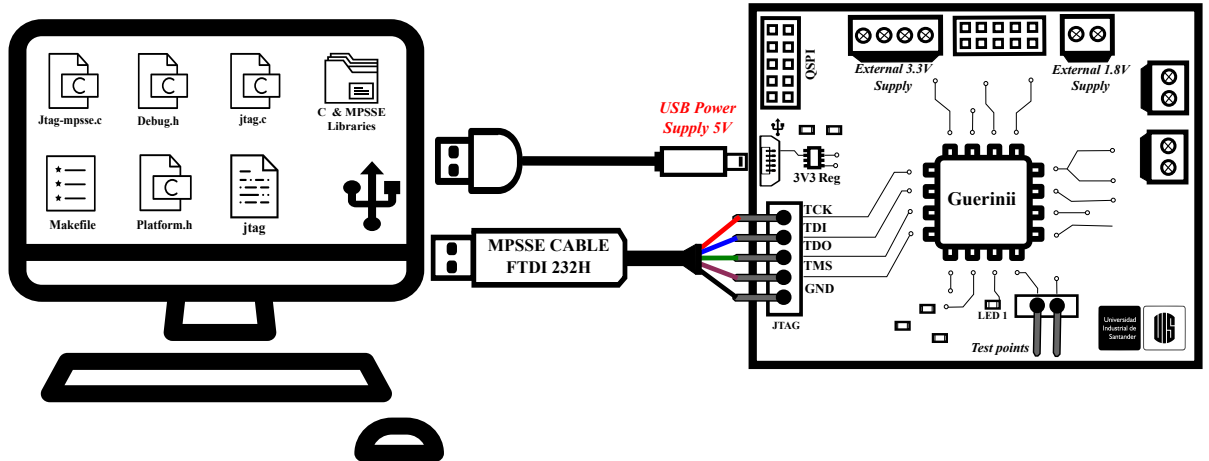
- Hardware:



- Computer.
- An FT232H single channel Hi-Speed-MPSSE cable to multipurpose UART/FIFO IC.
- A micro USB to B-type USB cable.
- Variable power supplies (\*optional)
- A breadboard.
- Test board for the microcontroller.
  - \* B-type micro USB connector.
  - \* Pin headers for JTAG ports.
  - \* 3.3V regulator.
  - \* LED.
  - \* Test points.
  - \* Electronic terminals.
  - \* Pin female headers.
- Software:
  - MPSSE and gcc libraries.
  - Software interface presented in Chapter 4.

Figure 45 illustrates the connection made to perform the experimental tests. As this thesis has mentioned previously, in the computer, the MPSSE libraries must be installed to connect/use the MPSSE products. Moreover, gcc libraries are required to work with the C language. The MPSSE cable is connected by USB to the computer, and the other end contains multiple outputs that are connected to the test board. These outputs provide the JTAG signals, and they are attached

Figure 45. Illustration of the connection set-up to test the microcontroller.



to test board JTAG ports. In the test board, the JTAG ports are connected to the pads of the IC. To supply the board and the microcontroller, there are two options. First, the board can be powered by using a micro USB to USB cable to provide a 5V power supply. The microcontroller must be supplied with 3.3V. Hence, a 3.3V regulator IC is connected to the B-type USB connector inputs. Thus, the output of the regulator powers the board and the chip. Second, the test board has electronic terminals for external supply. In this option, the regulator does not act and the voltage is supplied directly by a power supply that provides 3.3V and 1.8V. The test board also contains pin female headers to connect a QSPI memory to provide instructions to the microcontroller. In addition, pin headers and a led are connected in the platform to check controls made to the GPIOs.

**4.2.2. Planning and measurement results** This subsection presents the measurement plan. Here, the document describes what was measured, how it was measured and the obtained measurement results. Table 9 describes the order of the tests to be applied and a description of them. These tests are se-

Table 9. Planning of the experimental tests.

| # | Test          | Tested system | Description                                |
|---|---------------|---------------|--------------------------------------------|
| 1 | Bypass        | JTAG TAP      | Runs a bypass inst. to prove connectivity  |
| 2 | GPIOs Control | JTAG BS       | Turns on/off GPIOs using BS                |
| 3 | Halt Core     | Debug Module  | Halts a blinking program provided by QSPI. |
| 4 | Reset Core    | Debug Module  | Restarts the blinking program.             |
| 5 | DM Read       | Debug Module  | Reads a DM register previously written.    |
| 6 | AHB Write     | Debug Module  | Turn on/off a GPIO using the system bus    |
| 7 | AHB Read      | Debug Module  | Reads a register previously written by AHB |

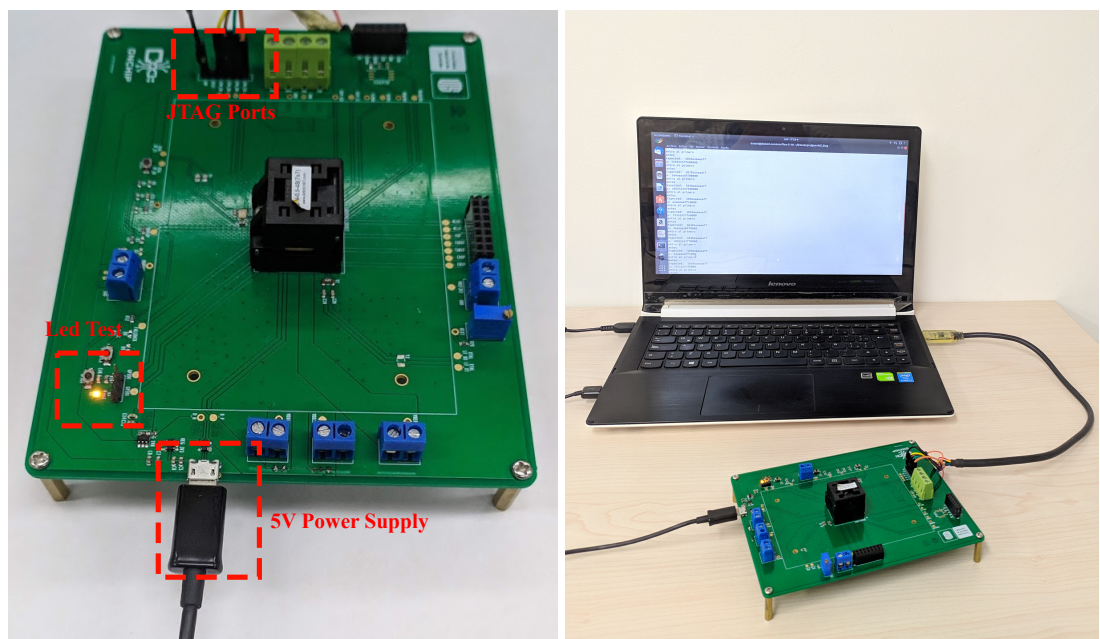
Table 10. Approval indicators of the tests.

| # | Test          | Approval Indicator                                         |
|---|---------------|------------------------------------------------------------|
| 1 | Bypass        | Software Interface must indicate: Bypass successful.       |
| 2 | GPIOs Control | Test led in the board must turn on/off.                    |
| 3 | Halt Core     | The test led must keep on or off.                          |
| 4 | Reset Core    | The led is stopped momentarily and the blinking continues. |
| 5 | DM Read       | Software interface displays the same programmed value.     |
| 6 | AHB Write     | Test led in the board must turn on/off.                    |
| 7 | AHB Read      | Software interface displays the same programmed value.     |

lected due to they check all the basic operations of the debugger. The approval indicators are indicated in Table 10. The first system to be tested is the JTAG TAP using a *Bypass* instruction to prove the connectivity with the chip.

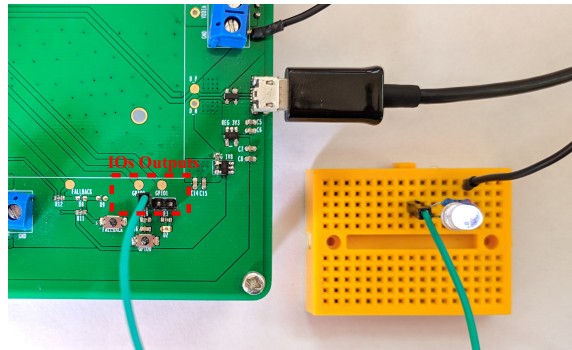
The second test is the GPIOs Control which consists of operating the boundary-scan to turn on/off the microcontroller IOs ports. The presented microcontroller was fabricated with two IO ports. Hence, there are 12 signals that are controlled by the boundary-scan (six for each IO, see Figure 12). To turn on the first IO, we put the following configuration in the  $IO_1$  using a boundary-scan instruction:  $ds = 0$ ,  $pue = 0$ ,  $oe = 1$ ,  $ie = 0$ ,  $o = 1$ ,  $i = don't\ care$ . This configuration enables the IO output and sends a high in the first IO. The boundary-scan instruction that

Figure 46. IOs control by the boundary-scan register.

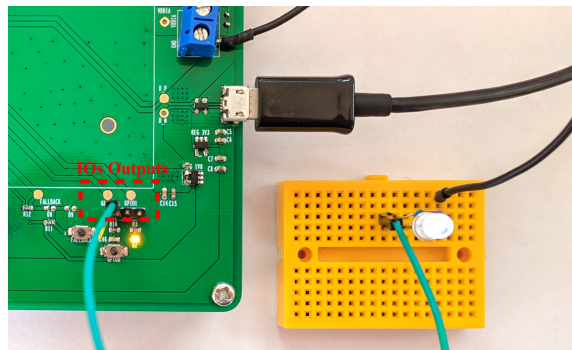


(a) IO put on high using boundary-scan. The (b) Test connection: Test board connected to the led is connected to the IO pad. the PC by MPSSE cable and USB power.

Figure 47. Photographs of the board in the execution of the blinking generated by boundary-scan.

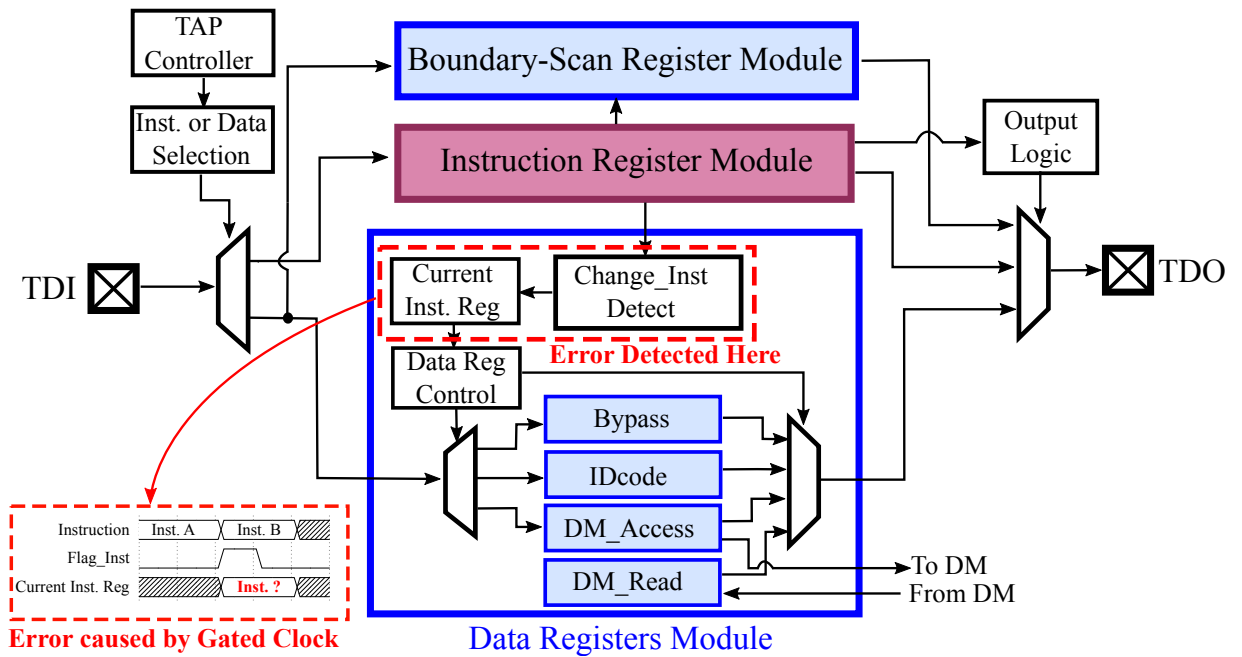


(a)  $IO_1$  put on high using boundary-scan. An additional test led inserted in a breadboard is connected to the pin header of the  $IO_2$  output.



(b) Blinking execution capture. Test led in the board and test led in the breadboard are turning on/off consecutively.

Figure 48. Simplified block diagram of the JTAG structure. A deepening in the data registers module is presented showing where the error is detected.



allows injecting signals in the IOs is *Extest*. Therefore, the first IO can be turned on by applying the value 0x5 (00101 in binary) in five *TCK* cycles using an *Extest* instruction. The function *EXTEST\_INST(jtag, 0x5, 5)* (see Figure 40) performs the mentioned process. Figure 46 shows images of the connection of this test. Figure 46(a) indicates the JTAG connection in the board, the 5V power supply, and the test led connected to the first IO (which is on). Figure 46(b) shows an image of the board connected to a PC where the program is run. To turn on the second GPIO, the 0x5 is sent in 11 *TCK* cycles using the *EXTEST\_INST(jtag, 0x5, 11)* function. A blinking program is described by using a combination of *EXTEST\_INST(jtag, 0x5, 5)* and *EXTEST\_INST(jtag, 0x5, 11)* functions to end the test. Figure 47 presents the result of this execution.

Test number 3 named Halt Core consist of stopping a blinking program pro-

vided by a QSPI memory. The memory contains instructions previously loaded which are read by the core. The execution of the test should keep static the led on or off. To perform this operation, a *Debug\_Write* instructions with a *Halt\_Core* order is described in a program code. However, the execution of this program did not stop the blinking. After several attempts to execute the program, the execution of the core could not be stopped.

The tests applied to the debug module failed. The codes executed could not trigger any of the operations of the debug module. This forced a thorough review of the software implemented and the design sent to be manufactured. In this review, we perform post-synthesis and place & route simulations of the full chip to found the error. Simulating the same process performed in the silicon implementation, we observed that the connection to the debug module was indeed broken. None of the instructions to the debug module worked. Tracking the process performed to write to the DM, we found the error that impeded the correct communication between the JTAG and the debug module. The error is described below:

Figure 48 illustrates a simplified block diagram of the JTAG. In this figure, the operational structure of the data registers is deepened. When the instruction register changes, a flag is activated in the *Data Registers Module* and this instruction is captured and loaded in a register. The error is detected in this part of the module as Figure 48 indicates. The found problem is that the change in the instruction and the enabling flag that triggered the capture of the *Current Inst. Reg*, occur at the same time. In typical simulation, this error did not occur. However, in the post-synthesis simulations of place & route, the simultaneous change makes the value captured by the register was not the desired one. The error is caused by a gated clock described in HDL. The boundary-scan was not affected by this error

due to this shift-register is independent of the data registers module.

**4.2.3. Experimental results and discussion** The experimental tests presented in the previous subsection are applied to a version of the platform fabricated in mid-2018. In this tapeout, some factors prevented obtaining a satisfactory pre-silicon verification process:

- Verification of the platform was always done at HDL level, both stand-alone and with the full chip.
- A post-gate verification with the complete chip was part of the microcontroller design plan. However, there was not a synthesizable version of the complete chip to do the verification within the timeline specified by the fabrication company.
- Many delays with the sign off occurred due to the research group was performing this process for the first time digitally.

In this microcontroller tapeout, fabrication time was an obstacle to perform all the relevant simulations (synthesis and place & route) that guarantee the operation of the design at a logic gates level. The functional verification process presented in Chapter 3 checked the correct operation of the design. Nonetheless, the error was not detected in these simulations since this process should also be performed at the gate level.

The proposed experimental tests could not be completed due to the error presented. Not performing the post-synthesis and place & route simulations led to a serious failure within the JTAG. Unfortunately, this error avoids communication with the debug module who performs the operation functions of the platform. The



control of the IOs, which is a function used to implement different control applications, can only be executed through the boundary-scan. Only the boundary-scan tests were successful due to their independence from the data registers module as Figure 48 presents.

The error found in the silicon implementation is corrected and the platform is redesigned. The register with a gated clock is replaced by a register in TCK domain with an enable signal. An enhanced version implemented in an FPGA is presented in the next section. The platform integrated within the microcontroller is tested using an FPGA and the software interface described in Chapter 4.

### 4.3. FPGA IMPLEMENTATION RESULTS

The full SoC is synthesized and implemented in an Artix-7 35T Arty FPGA to test the debugging platform and the software interface working together (44). This section presents different tests performed to the debugging platform using the libraries described in Chapter 4. The results to be exposed are obtained by using an integrated logic analyzer (ILA) that can be added to the implementation in FPGA. The ILA allows capturing the operation of different signals of the design.

**Device Initialization and Debug Module Writing** Figure 49 presents a code example where the initialization of the device is described. In this code, a pair of instructions for initializing and recognizing the device are programmed, followed by an instruction to halt the core. These instructions are translated by the debug adapter (the USB Hi-Speed-MPSSE cable in this case) to generate the JTAG protocol signaling. Then, the debug adapter sends that signaling to the JTAG ports assigned in the FPGA. Figure 50 illustrates the operation captured by the ILA. The capture shows the initialization process: first, a recognition device stage that uses a bypass instruction; second, an IDCode instruction is consecutively executed to obtain the ID of the device. After these two stages, the program instructions are executed.

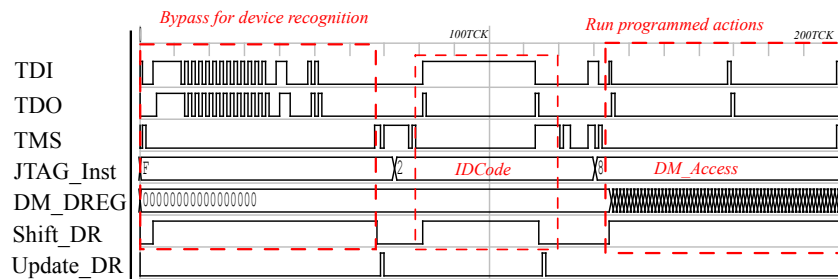
The halt core instruction programmed in the code shown in Figure 49 is also captured by the ILA. Figure 51 depicts the process where the data to be sent to the debug module are loaded in a data register of JTAG. Once the data is loaded, it is divided into three signals: data, address and operation. In the *Update\_DR* state of the TAP Controller, JTAG sends this information to the DM. The ILA has allowed observing the execution process by the program. Thereby, we can

Figure 49. Code description of the device initialization.

```
#include <stdio.h>
#include "debug.h" //debugging library
int main(int argc, char **argv) {
    JTAG *jtag; //Create a jtag handle

    if (!jtag = jtag_open()) //Detect and open device
        return -1;
    if ((n = jtag_enumerate(jtag)) <= 0) //Device recognition(optional)
        return -1;
    //Here the program starts
    Debug_write(jtag, Halt_Core, DMI_CONTROL_CORE, 1);
}
```

Figure 50. Measured waveforms of the initialization of the device. These signals are captured by the ILA.



check that the software is able to make the debugging platform to perform actions. The software interface must display the actions that are being executed in the program. Figure 52 shows the information displayed by the software interface in the Linux console. The console displays the instructions performed, and the initialization process and the transaction to the debug module are indicated.

Figure 53 shows another example which describes a sequence to halt, reset, and resume the core. To monitor this program, the ILA is integrated into the debug module. Figure 54 displays the execution of this program in the debug module. The debug module is enabled by the first *Debug\_write* instruction. Later, the halt core order is written in the *Control\_Core* register. This makes the halt signal to be

Figure 51. Measured waveforms of the data sending to the DM.

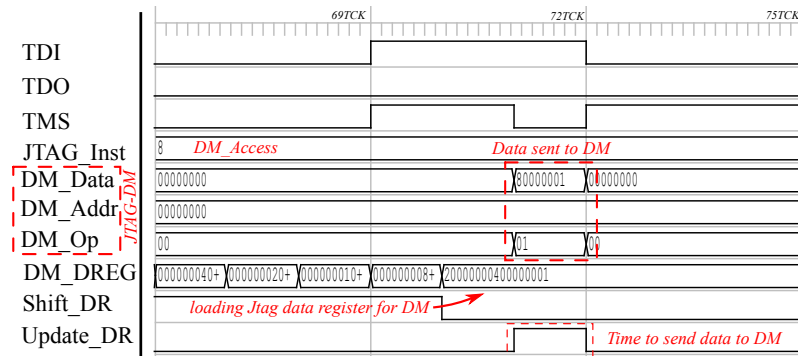


Figure 52. Information displayed in Linux console.

```
daniel@daniel-Lenovo-Flex-2-14:~/Descargas/JTAG_Software_Interface$ ./jtag
-----Initialization Process:IDcode and Bypass Test -----
Expected Value in Bypass Test: 5038aaaaaaff
Obtained Value: 5038aaaaaaff
jtag: device 0 idcode 10000001 unknown IR size
*****Initialization process finished*****
Debug Module Connection --> Data: 80000001 ; Address: 0; Op: 2
```

set high, which stops the pc of the core. Then, the reset command is written in the *Control\_Core* register and activates the reset signal to restart the core. Finally, the core is resumed. *DMI\_Control\_Core*, *Resume\_Core*, *Halt\_Core*, *Reset\_Core* parameters are defined in *debug.h*. The performed actions are displayed in the Linux console as shown in Figure 55, where the four debug module transfers are indicated.

Figure 57. Measured waveforms of the AHB system bus transaction.

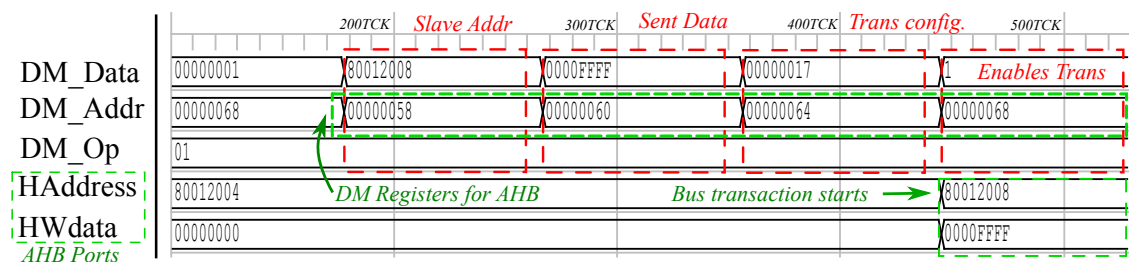
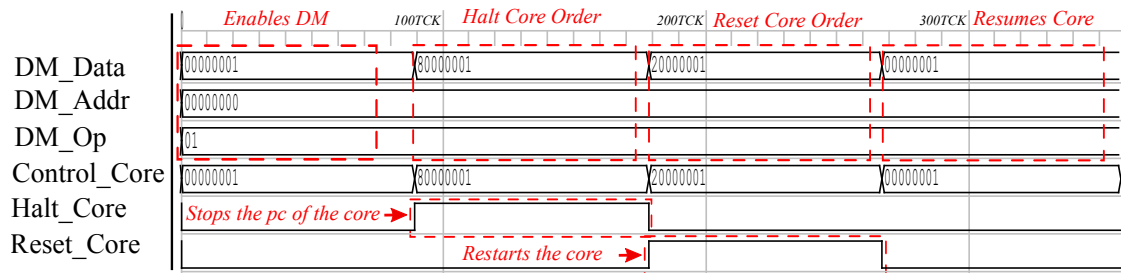


Figure 53. Code example to send data to the DM.

```
#include <stdio.h>
#include "debug.h" //debugging library
int main(int argc, char **argv) {
    JTAG *jtag; //Create a jtag handle

    if (!jtag = jtag_open()) //Detect and open device
        return -1;
    //Here the program starts
    Debug_write(jtag, Resume_Core, DMI_CONTROL_CORE, 1);
    Debug_write(jtag, Halt_Core, DMI_CONTROL_CORE, 1);
    Debug_write(jtag, Reset_Core, DMI_CONTROL_CORE, 1);
    Debug_write(jtag, Resume_Core, DMI_CONTROL_CORE, 1);
}
```

Figure 54. Measured waveforms of the DM writing: halt and reset orders.



**System Bus Access** The system bus access allows communicating to any slave connected to the bus. Figure 56 describes a program where two accesses to the system bus are programmed. The ILA is put within the DM to monitor the AHB ports data and address, and the data sent from JTAG. Figure 57 presents the signals captured by the ILA. Four steps are executed for each AHB transaction. DM writings are performed to load the transaction data in DM registers for AHB. The slave address, the data to be sent, and other AHB configurations are loaded in these registers. Once the information is loaded, the four DM writing enables the transaction and the bus transaction starts. All actions performed in this program are indicated in the console as Figure 58 illustrates. In this case, we can observe

Figure 55. Program execution displayed in Linux console for a DM writing.

```
daniel@daniel-Lenovo-Flex-2-14:~/Descargas/JTAG_Software_Interface$ ./jtag
-----Initialization Process:IDcode and Bypass Test -----
Expected Value in Bypass Test: 5038aaaaaaff
Obtained Value: 5038aaaaaaff
jtag: device 0 idcode 10000001 unknown IR size
*****
Debug Module Connection --> Data: 1 ; Address: 0; Op: 0
Debug Module Connection --> Data: 80000001 ; Address: 0; Op: 2
Debug Module Connection --> Data: 20000001 ; Address: 0; Op: 0
Debug Module Connection --> Data: 1 ; Address: 0; Op: 0
```

Figure 56. Code example of an AHB system bus transaction.

```
#include <stdio.h>
#include "debug.h" //debugging library
int main(int argc, char **argv) {
    JTAG *jtag; //Create a jtag handle

    if (!jtag = jtag_open()) //Detect and open device
        return -1;
    //Here the program starts
    AHB_Access(jtag, 0x80012004, 0x0, true);
    AHB_Access(jtag, 0x80012008, 0xFFFF, true);
}
```

indications about system bus transactions where the register address, the data and the operation performed are specified. Each system bus transaction involves debug module connections that are also specified in the console.

**Debug Module Status** The *DM\_Status* register in JTAG allows knowing the operation state of the debug module. Thereby, we can check if the programmed operations are executed successfully. Besides having a record and control of the operations that have been executed. To access to the *DM\_Status* register by the software interface, we use the function *DM\_info()*. For instance, if an instruction *DM\_info(jtag)* is added to the end of the code shown in Figure 49, the Linux

Figure 58. Program execution about AHB transactions displayed in Linux console.

```
daniel@daniel-Lenovo-Flex-2-14:~/Descargas/JTAG_Software_Interface$ ./jtag
-----Initialization Process:IDcode and Bypass Test -----
Expected Value in Bypass Test: 5038aaaaaff
Obtained Value: 5038aaaaaff
jtag: device 0 idcode 10000001 unknown IR size
*****
System Bus (AHB) Transaction --> Reg Address: 80012004; Data: 0; Op: WRITE
Debug Module Connection --> Data: 1 ; Address: 0; Op: 0
Debug Module Connection --> Data: 80012004 ; Address: 58; Op: 2
Debug Module Connection --> Data: 0 ; Address: 60; Op: 0
Debug Module Connection --> Data: 17 ; Address: 64; Op: 0
Debug Module Connection --> Data: 1 ; Address: 68; Op: 0
System Bus (AHB) Transaction --> Reg Address: 80012008; Data: ffff; Op: WRITE
Debug Module Connection --> Data: 1 ; Address: 0; Op: 0
Debug Module Connection --> Data: 80012008 ; Address: 58; Op: 2
Debug Module Connection --> Data: ffff ; Address: 60; Op: 0
Debug Module Connection --> Data: 17 ; Address: 64; Op: 0
Debug Module Connection --> Data: 1 ; Address: 68; Op: 0
```

Main Operations

Involved Operations

Figure 59. Debug module status printed in Linux console.

```
daniel@daniel-Lenovo-Flex-2-14:~/Descargas/JTAG_Software_Interface$ ./jtag
-----Initialization Process:IDcode and Bypass Test -----
Expected Value in Bypass Test: 5038aaaaaff
Obtained Value: 5038aaaaaff
jtag: device 0 idcode 10000001 unknown IR size
*****
Debug Module Connection --> Data: 80000001 ; Address: 0; Op: 2

-----DM STATUS-----
Debug Module Version: 1
Debug Module State: ENABLED
Halted Core: YES
Reset Core: DISABLED
PC Mode: DISABLED
Programmed Program Buffer Cells: 0
Performed AHB Transfers: 0
Total Access to DM: 1
Enabled monitoring lines: 0
AHB read pending for extracting: NO
-----END STATUS-----
```

Figure 60. Code example of using the function *DM\_info()*.

```
#include <stdio.h>
#include "debug.h" //debugging library
int main(int argc, char **argv) {
    JTAG *jtag; //Create a jtag handle

    if (!(jtag = jtag_open())) //Detect and open device
        return -1;
    //Here the program starts
    Debug_write(jtag, Halt_Core, DMI_CONTROL_CORE, 1); -Inst. 1
    DM_info(jtag); -Inst. 2
    Debug_write(jtag, Resume_Core, DMI_CONTROL_CORE, 1); -Inst. 3
    AHB_Access(jtag, 0x80012004, 0x0, true); -Inst. 4
    AHB_Access(jtag, 0x80012008, 0xFFFF, true); -Inst. 5
    DM_info(jtag); -Inst. 6
}
```

console displays a debug module status as shown in Figure 59. The printed information in the console is associated with the data stored in the bits 0 to 27 of the *DM\_Status* register presented in Figure 9. The function extracts the data from the JTAG and executes an algorithm to separate the information of each register presented in Figure 9. Then, it organizes the information to be presented as depicted in Figure 59. Figure 60 presents an extended example where some programmed instructions are enumerated. The enumerated instructions are highlighted in Figure 61 which illustrates the execution of the program in the Linux console. Figure 61 shows how the function *DM\_info()* (instrs. 2 and 6) displays information according to the instructions that are being executed. For this instance, the states: "Debug Module State", "Halted Core", "Performed AHB Transfers" and "Total access to DM" contain the relevant information.



Figure 61. Execution of the program in Figure 60.

```

daniel@daniel-Lenovo-Flex-2-14:~/Descargas/JTAG_Software_Interface$ ./jtag
-----Initialization Process:IDcode and Bypass Test -----
Expected Value in Bypass Test: 5038aaaaaff
Obtained Value: 5038aaaaaff
jtag: device 0 idcode 10000001 unknown IR size
*****
Debug Module Connection --> Data: 80000001 ; Address: 0; Op: 2
Instr. 1

-----DM STATUS-----
Debug Module Version: 1
Debug Module State: ENABLED
Halted Core: YES
Reset Core: DISABLED
PC Mode: DISABLED
Programmed Program Buffer Cells: 0
Performed AHB Transfers: 0
Total Access to DM: 1
Enabled monitoring lines: 0
AHB read pending for extracting: NO
-----END STATUS-----
Instr. 2

Debug Module Connection --> Data: 1 ; Address: 0; Op: 0
Instr. 3
System Bus (AHB) Transaction --> Reg Address: 80012004; Data: 0; Op: WRITE
Debug Module Connection --> Data: 1 ; Address: 0; Op: 0
Debug Module Connection --> Data: 80012004 ; Address: 58; Op: 2
Debug Module Connection --> Data: 0 ; Address: 60; Op: 0
Instr. 4
Debug Module Connection --> Data: 17 ; Address: 64; Op: 0
Debug Module Connection --> Data: 1 ; Address: 68; Op: 0
System Bus (AHB) Transaction --> Reg Address: 80012008; Data: ffff; Op: WRITE
Debug Module Connection --> Data: 1 ; Address: 0; Op: 0
Debug Module Connection --> Data: 80012008 ; Address: 58; Op: 2
Debug Module Connection --> Data: ffff ; Address: 60; Op: 0
Instr. 5
Debug Module Connection --> Data: 17 ; Address: 64; Op: 0
Debug Module Connection --> Data: 1 ; Address: 68; Op: 0

-----DM STATUS-----
Debug Module Version: 1
Debug Module State: ENABLED
Halted Core: NO
Reset Core: DISABLED
PC Mode: DISABLED
Programmed Program Buffer Cells: 0
Performed AHB Transfers: 2
Total Access to DM: 12
Enabled monitoring lines: 0
AHB read pending for extracting: NO
-----END STATUS-----
Instr. 6

```

#### 4.4. SUMMARY

Table 11. Key Features of the proposed debugging platform.

| Key feature                            | Description                                                                                                 |
|----------------------------------------|-------------------------------------------------------------------------------------------------------------|
| Core and system bus run-control        | DM controls the core operation and performs transfers based on AMBA buses.                                  |
| Non-intrusive monitoring               | BM does not intervene in the operation of the SoC.                                                          |
| Flexible configuration                 | Multiple modes for analysis with filtered transaction detection.                                            |
| BM data storage is reduced to counters | Filters avoid to store large/undesired captured data.                                                       |
| Bus monitors are adaptable             | BM operation is independent of the bus protocol (for instance AMBA buses).                                  |
| Reusable debugging platform            | The debugging platform (DP) can be easily reused and adapted in other SoC.                                  |
| Easily expandable system               | The DP is scalable. It can be synthesized with configurable number of program buffer cells and filters/PCs. |

A debugging platform to exercise post-silicon tests of a full SoC has been presented in this master's thesis. The platform is capable to inject instructions, to capture data transfers and to perform analysis within a system containing a 32-bit processor, a system bus and a full set of peripherals. Multiple operation configurations for DM and BM can be set to perform defined debug tasks. The system supports a selection of specific data or ranges of data, together with specific addresses or ranges of addresses (either destination or source), along with the option to decide on reading or writing operations. The proposed system is flexible and easy to configure, and regarding its communication with the bus and the core, it also offers comprehensive observability and controllability for post-silicon testing. Tabla 11 summarizes the features of the proposed debugging platform

design presented in this master's thesis. The final design in section 5.3 presents the features mentioned in this Table. Moreover, a software interface is presented in Chapter 4 to operate the debugging platform. Some code examples are described and explained in order to provide an application guide for future software users.

In addition, a silicon implementation of the platform is also presented in this thesis. The platform integrated within a 32-bit microcontroller is fabricated in TSMC 0.18 $\mu$ m. The microcontroller had issues in the debug module connectivity due to an error not detected by not performing full SoC post-synthesis simulation. The error is corrected for the version shown in section 5.3 where an enhanced version of the platform is presented.

## 5. CONTRIBUTIONS AND FUTURE WORK

This chapter describes the contributions made by this research work and how the work opens the doors to new fields of research and development. In the contributions, scientific dissemination articles and undergraduate projects are presented. On the other hand, a description of future work is presented by indicating long and short term projects.

### 5.1. CONTRIBUTIONS

The research works developed in master's studies must contribute to the public knowledge of the academy and open doors for new research and development approaches. This section exposes the contributions of the presented thesis. First, Section 6.1.1 presents the published works in the elaboration of the research work, and second, Section 6.1.2 mentions a bachelor's project that is based on the work performed in this thesis.

#### 5.1.1. Related articles    **W. Ramirez** and E. Roa. **Post-Silicon Debugging Platform to Perform Behavioral and Performance Analyzes** (LASCAS 2019) (45)

Post-silicon debugging systems must offer run-control capability and visibility on complex SoC in order to detect/analyze errors and find possible design enhancements. This paper presents a scalable and reusable debugging platform for post-silicon validation. The platform is composed of a debug module with JTAG communication and a bus monitor. Relevant features are core control, system bus operation, and non-intrusive monitoring. Flexible filtering allows se-

lecting transfers of interest or performing general monitoring of the SoC. Captured data can be analyzed by performance counters that check executed and finished transfers, calculating their latency to detect such as deadlocks in the system. The debugging platform has been implemented as part of an SoC with a 32-bit RISC-V based core and multiple peripherals. This paper was presented and published by 2019 IEEE 10th Latin American Symposium on Circuits & Systems (LASCAS).

**W. Ramirez, H. Gomez and E. Roa. On UVM Reliability in Mixed-Signal Verification (LASCAS 2019) (46)**

During the last decade, Universal Verification Methodology (UVM) has become a popular standard test methodology for verification of intellectual property (VIP) within digital and mixed-signal systems. UVM prominent features include stimulus automation, I/O checking and code reuse. This paper analyzes the strengths and weaknesses of UVM along with measurements of reliability using a 32-bit LPDDR3 memory interface and a bandgap voltage reference. Simulation results indicate that reliability is limited by the complexity of the circuit under test and proper UVM setup to get considerable analog simulation coverage. For analog cases, UVM-AMS can render low reliability considering that a common practice in analog design is creating multiple testbenches according to the function/domain tested. VIP by itself should be used as a complement to traditional verification practices even when assuming access to a fully detailed UVM-AMS VIP. This paper was presented and published by 2019 IEEE 10th Latin American Symposium on Circuits & Systems (LASCAS).

**5.1.2. Bachelor's Projects** This master's thesis has opened the way to a new research field for Onchip group and UIS. This field is the debugging of integrated systems. As proof of this, an undergraduate project that is focused on one of the features of the debugging system is currently under development. The project is described next:

Marco Sarmiento, “**Técnica de depuración para el control de procesadores RISC-V de 32 bits**”, Universidad Industrial de Santander, Colombia.

**Description:** The main goal of this project is to describe a debugging technique for the control of a 32-bit RISC-V processor. The project is focused on deepening and improving features of a specific operation of the debugging system presented in this master's thesis. The mentioned operation is the core control. The description will be made qualitatively and will be tested using a hardware description language. The director of this bachelor's project is Wilmer Ramirez (author of this master's thesis) and the co-director is Elkim Roa.

## **5.2. FUTURE WORK**

This section mentions the work to be done after the development of the presented debugging system. Here, short-term work is mentioned which is called "Amazilia" and long-term objectives for the research in integrated systems are exposed.

**SHORT-TERM WORK** The integrated systems research group Onchip from Universidad Industrial de Santander developed in 2018 the project named Guerinii, which consist of a 32-bit microcontroller RISC-V with peripherals for IoT and security instances. In this microcontroller, the presented debugging platform was

implemented. In 2019, Onchip has a new project which is named Amazillia. One of the new features of Amazilia is to improve the debugging system focusing on the part of the complete control of the processor. To achieve this, a control interface to access to the control and status registers (CSR) of the core is required. Moreover, new techniques for core controlling seek to be implemented.

**LONG-TERM WORK** Onchip group seeks to continue with the research in debugging systems. Below are some long-term goals:

- To implement a real-time monitoring and control of digital and analog circuits for managing the functional logic and the power consumption of an SoC.
- To implement an optimal control on/off of the systems that integrate an SoC in order to overcome the problem generated by the dark silicon (47).
- To ingrate USB protocol to the debugging system.

## REFERENCES

- [1] ARM, “AMBA AXI and ACE protocol specification,” 2011, pp. 1–121.
- [2] “IEEE Standard for Test Access Port and Boundary-Scan Architecture,” *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)*, pp. 1–444, May 2013.
- [3] J. P. Linda E. M. Brackenbury, Luis A. Plana, “System-on-Chip Design and Implementation,” in *IEEE Transactions On Education*, August 2009.
- [4] A. Hazra, S. Goyal, P. Dasgupta, and A. Pal, “Formal Verification of Architectural Power Intent,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 1, pp. 78–91, Jan 2013.
- [5] B. C. Lim and M. Horowitz, “An Analog Model Template Library: Simplifying Chip-Level, Mixed-Signal Design Verification,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 1, pp. 193–204, Jan 2019.
- [6] C. Liang, G. Zhong, S. Huang, and B. Xia, “Uvm-ams based sub-system verification of wireless power receiver soc,” in *2014 12th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, Oct 2014, pp. 1–3.
- [7] K. Salah, “A UVM-Based Smart Functional Verification Platform: Concepts, pros, cons, and opportunities,” in *2014 9th International Design and Test Symposium (IDT)*, Dec 2014, pp. 94–99.



- [8] C. Kuznik and W. Muller, "Aspect enhanced functional coverage driven verification in the systemc hdvl," in *2011 International SoC Design Conference*, Nov 2011, pp. 154–157.
- [9] K. Datta and P. P. Das, "Assertion based verification using hdvl," in *17th International Conference on VLSI Design. Proceedings.*, 2004, pp. 319–325.
- [10] H. B. Carten and S. G. Hemmady, *Metric Driver Design Verification*. Springer Science + Bussiness Media, 2007.
- [11] A. Adir, S. Copty, S. Landa, A. Nahir, G. Shurek, A. Ziv, C. Meissner, and J. Schumann, "A Unified Methodology for Pre-Silicon Verification and Post-Silicon Validation," in *2011 Design, Automation Test in Europe*, March 2011, pp. 1–6.
- [12] U. T. Ltd. (2015) Performance monitoring using ultrasoc. [Online]. Available: <https://www.ultrasoc.com/downloads>
- [13] ——. Soc Debug. [Online]. Available: <https://www.ultrasoc.com/applications/soc-development>
- [14] K. Lee, S. Liang, and A. Su, "A Low-Cost SoC Debug Platform Based on On-Chip Test Architectures," in *2009 IEEE International SOC Conference (SOCC)*, Sept 2009, pp. 161–164.
- [15] M. H. Neishaburi and Z. Zilic, "An Enhanced Debug-aware Network Interface for Network-on-Chip," in *Thirteenth International Symposium on Quality Electronic Design (ISQED)*, March 2012, pp. 709–716.
- [16] —, "A Distributed AXI-based Platform for Post-silicon Validation," in *29th VLSI Test Symposium*, May 2011, pp. 8–13.

- [17] H. F. Ko, A. B. Kinsman, and N. Nicolici, "Design-for-Debug Architecture for Distributed Embedded Logic Analysis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 8, pp. 1380–1393, Aug 2011.
- [18] H. m. Kyung et al., "Design and Implementation of Performance Analysis Unit (PAU) for AXI-based Multi-core System on a Chip (soc)," *Microprocess. Microsyst.*, vol. 34, pp. 102–116, Jun 2010.
- [19] ARM, *AMBA AXI<sup>TM</sup> and ACE<sup>TM</sup> Protocol Specification*, 2011.
- [20] U. of California in Berkeley, "Chisel - Constructing Hardware in a Scala Embedded Language," [Online] Available: <https://chisel.eecs.berkeley.edu/>.
- [21] SiFive, *RISC-V External Debug Support 0.13*, 2018.
- [22] L. Pierce and S. Tragoudas, "Multi-level secure jtag architecture," in *2011 IEEE 17th International On-Line Testing Symposium*, July 2011, pp. 208–209.
- [23] C. Clark, "Anti-tamper jtag tap design enables drm to jtag registers and p1687 on-chip instruments," in *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, June 2010, pp. 19–24.
- [24] T. Newsome, "RISC-V External Debug Support," SiFive Inc., Apr 2017, This draft specification will change before being accepted as standard, so implementations made to this draft specification will likely not conform to the future standard. [Online]. Available: <https://dev.sifive.com/documentation/risc-v-external-debug-support/>
- [25] ARM, *AMBA Specification (Rev 2.0)*, 1999.

- [26] "IEEE Standard for Universal Verification Methodology Language Reference Manual," *IEEE Std 1800.2-2017*, pp. 1–472, May 2017.
- [27] Y. N. Yun, J. B. Kim, N. D. Kim, and B. Min, "Beyond UVM for Practical soc Verification," in *2011 International SoC Design Conference*, Nov 2011, pp. 158–162.
- [28] C. Liu, Z. Xie, J. Su, Q. Liu, and X. Wang, "An Intelligent and Reusable Verification Platform Based on UVM for RFID Digital Baseband," in *2014 IEEE International Conference on Electron Devices and Solid-State Circuits*, June 2014, pp. 1–2.
- [29] V. Academy, "Introduction to uvm," [Online] Available: <https://verificationacademy.com/sessions/introduction-uvm>.
- [30] Open SoC Debug. Open SoC Debug Specification. [Online]. Available: [https://opensocdebug.readthedocs.io/en/latest/02\\_spec/index.html](https://opensocdebug.readthedocs.io/en/latest/02_spec/index.html)
- [31] O. S. Debug. Open soc debug primer. [Online]. Available: [https://opensocdebug.readthedocs.io/en/latest/01\\_overview/getting-started/index.html](https://opensocdebug.readthedocs.io/en/latest/01_overview/getting-started/index.html)
- [32] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with gdb - the gnu Source-Level Debugger. [Online]. Available: <https://www.gnu.org/software/gdb/documentation>
- [33] Free Software Foundation, Inc. GNU. Gdb: The GNU Project Debugger. [Online]. Available: <https://www.gnu.org/software/gdb/>
- [34] Open SoC Debug. Open soc debug - building blocks for your soc debug infrastructure. [Online]. Available: <https://opensocdebug.org>

- [35] LowRisc. A fully open-sourced, linux-capable, system-on-a-chip. [Online]. Available: <https://www.lowrisc.org/>
- [36] OpTiMSoC. Optimsoc. [Online]. Available: <https://optimsoc.org/>
- [37] Philipp Wagne and Stefan Wallentowitz. The Generic Logic Interfacing Project . [Online]. Available: <https://www.glip.io>
- [38] Dominic Rath , *Open On-Chip Debugger - Design and Implementation of an On-Chip Debug Solution for Embedded Target Systems based on the ARM7 and ARM9 Family*. University of Applied Sciences Augsburg - Department of Computer Science, 2005.
- [39] The OpenOCD Project et al. , *Open On-Chip Debugger:OpenOCD User's Guide*, 2017.
- [40] Eddie Ashfield et al. ARM Ltd. Serial Wire Debug and the CoreSight Debug and Trace Architecture . [Online]. Available: [https://www.arm.com/files/pdf/Serial\\_Wire\\_Debug.pdf](https://www.arm.com/files/pdf/Serial_Wire_Debug.pdf)
- [41] F. T. D. I. LTD, “USB Hi-Speed - MPSSE Cables.” [Online]. Available: <https://www.ftdichip.com/Products/Cables/USBMPSSSE.htm>
- [42] H. Yi, S. Park, and S. Kundu, “A Design-for-Debug (DfD) for NoC-Based SoC Debugging via NoC,” in *2008 17th Asian Test Symposium*, Nov 2008, pp. 289–294.
- [43] Europractice. Europractice-ic service. [Online]. Available: <http://europractice-ic.com/>
- [44] Xilinx, “Artix-7 35T Arty FPGA Evaluation Kit.” [Online]. Available: <https://www.xilinx.com/products/boards-and-kits/artly.html#documentation>

- [45] **W. Ramirez** and E. Roa, "Post-Silicon Debugging Platform with Bus Monitoring Capability to Perform Behavioral and Performance Analyses," in *2019 IEEE 10th Latin American Symposium on Circuits Systems (LASCAS)*, Feb 2019, pp. 81–84.
- [46] **W. Ramirez**, H. Gomez, and E. Roa, "On UVM Reliability in Mixed-Signal Verification," in *2019 IEEE 10th Latin American Symposium on Circuits Systems (LASCAS)*, Feb 2019, pp. 233–236.
- [47] Edited by Ali R. Hurson, Hamid Sarbazi-Azad , *Advances in Computers, Dark Silicon and Future On-chip Systems*. Academic Press, 2018, vol. 110.

## BIBLIOGRAPHY

ACADEMY V. (2012). Introduction to UVM, [Online] Available:  
<https://verificationacademy.com/sessions/introduction-uvm>.

A. Adir, S. Coptly, S. Landa, A. Nahir, G. Shurek, A. Ziv, C. Meissner, and A. Hazra, S. Goyal, P. Dasgupta, and A. Pal, "Formal Verification of Architectural Power Intent," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 21, no. 1, pp. 78–91, Jan 2013.

ARM, AMBA AXI TM and ACETM Protocol Specification, 2011.

ARM, AMBA Specification (Rev 2.0), 1999.

ARM, "AMBA AXI and ACE protocol specification," 2011, pp. 1–121.

B. C. Lim and M. Horowitz, "An Analog Model Template Library: Simplifying Chip-Level, Mixed-Signal Design Verification," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 27, no. 1, pp. 193–204, Jan 2019.

C. Clark, "Anti-tamper jtag tap design enables drm to jtag registers and p1687 on-chip instruments," in 2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), June 2010, pp. 19–24.

C. Kuznik and W. Muller, "Aspect enhanced functional coverage driven verification in the systemc hdvl," in 2011 International SoC Design Conference, Nov 2011, pp. 154–157.

C. Liang, G. Zhong, S. Huang, and B. Xia, "Uvm-ams based sub-system verification of wireless power receiver soc," in 2014 12th IEEE International Conference

on Solid-State and Integrated Circuit Technology (ICSICT), Oct 2014, pp. 1–3.

C. Liu, Z. Xie, J. Su, Q. Liu, and X. Wang, “An Intelligent and Reusable Verification Platform Based on UVM for RFID Digital Baseband,” in 2014 IEEE International Conference on Electron Devices and Solid-State Circuits, June 2014, pp. 1–2.

Dominic Rath, Open On-Chip Debugger - Design and Implementation of an On-Chip Debug Solution for Embedded Target Systems based on the ARM7 and ARM9 Family. University of Applied Sciences Augsburg - Department of Computer Science, 2005.

Eddie Ashfield et al. ARM Ltd. Serial Wire Debug and the CoreSight Debug and Trace Architecture. [Online]. Available: <https://www.arm.com/files/pdf/SerialWireDebug.pdf>

Edited by Ali R. Hurson, Hamid Sarbazi-Azad, Advances in Computers, Dark Silicon and Future On-chip Systems. Academic Press, 2018, vol. 110. Europractice. Europractice-ic service. [Online]. Available: <http://europractice-ic.com/>

F. T. D. I. LTD, “USB Hi-Speed - MPSSE Cables.” [Online]. Available: <https://www.ftdichip.com/Products/Cables/USBMPSSE.htm>

Free Software Foundation, Inc. GNU. Gdb: The GNU Project Debugger. [Online]. Available: <https://www.gnu.org/software/gdb/>

H. B. Carten and S. G. Hemmady, Metric Driver Design Verification. Springer Science + Business Media, 2007.

H. F. Ko, A. B. Kinsman, and N. Nicolici, “Design-for-Debug Architecture for Distributed Embedded Logic Analysis,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 19, no. 8, pp. 1380–1393, Aug 2011.

- H. m. Kyung et al., "Design and Implementation of Performance Analysis Unit (PAU) for AXI-based Multi-core System on a Chip (soc)," *Microprocess. Microsyst.*, vol. 34, pp. 102–116, Jun 2010.
- H. Yi, S. Park, and S. Kundu, "A Design-for-Debug (DfD) for NoC-Based SoC Debugging via NoC," in *2008 17th Asian Test Symposium*, Nov 2008, pp. 289–294.
- IEEE, "IEEE Standard for Test Access Port and Boundary-Scan Architecture," *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)*, pp. 1–444, May 2013.
- IEEE, "IEEE Standard for Universal Verification Methodology Language Reference Manual," *IEEE Std 1800.2-2017*, pp. 1–472, May 2017.
- J. P. Linda E. M. Brackenbury, Luis A. Plana, "System-on-Chip Design and Implementation," in *IEEE Transactions On Education*, August 2009.
- J. Schumann, "A Unified Methodology for Pre-Silicon Verification and Post-Silicon Validation," in *2011 Design, Automation Test in Europe*, March 2011, pp. 1–6.
- K. Datta and P. P. Das, "Assertion based verification using hdlv," in *17th International Conference on VLSI Design. Proceedings.*, 2004, pp. 319–325.
- K. Lee, S. Liang, and A. Su, "A Low-Cost SoC Debug Platform Based on On-Chip Test Architectures," in *2009 IEEE International SOC Conference (SOCC)*, Sept 2009, pp. 161–164.
- K. Salah, "A UVM-Based Smart Functional Verification Platform: Concepts, pros, cons, and opportunities," in *2014 9th International Design and Test Symposium (IDT)*, Dec 2014, pp. 94–99.
- L. Pierce and S. Tragoudas, "Multi-level secure jtag architecture," in *2011 IEEE*



17th International On-Line Testing Symposium, July 2011, pp. 208– 209.

LowRisc. A fully open-sourced, linux-capable, system-on-a-chip. [Online]. Available: <https://www.lowrisc.org/>

M. H. Neishaburi and Z. Zilic, “A Distributed AXI-based Platform for Post-silicon Validation,” in 29th VLSI Test Symposium, May 2011, pp. 8–13.

M. H. Neishaburi and Z. Zilic, “An Enhanced Debug-aware Network Interface for Network-on-Chip,” in Thirteenth International Symposium on Quality Electronic Design (ISQED), March 2012, pp. 709–716.

O. S. Debug. Open soc debug primer. [Online]. Available: [https://opensocdebug.readthedocs.io/en/latest/01overview/getting started/index.html](https://opensocdebug.readthedocs.io/en/latest/01overview/getting%20started/index.html)

Open SoC Debug. Open SoC Debug Specification. [Online]. Available: [https://opensocdebug.readthedocs.io/en/latest/02 spec/index.html](https://opensocdebug.readthedocs.io/en/latest/02%20spec/index.html)

OpTiMSoC. Optimsoc. [Online]. Available: <https://optimsoc.org/>

Philipp Wagne and Stefan Wallentowitz. The Generic Logic Interfacing Project . [Online]. Available: <https://www.glip.io>

Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with gdb - the gnu Source-Level Debugger. [Online]. Available: <https://www.gnu.org/software/gdb/documentation>

SiFive, RISC-V External Debug Support 0.13, 2018.

T. Newsome, “RISC-V External Debug Support,” SiFive Inc., Apr 2017, This draft specification will change before being accepted as standard, so implementations made to this draft specification will likely not conform to the future standard.

[Online]. Available: <https://dev.sifive.com/documentation/risc-v-external-debug-support/>

The OpenOCD Project et al. ,Open On-Chip Debugger:OpenOCD User's Guide, 2017.

U. of California in Berkeley, "Chisel - Constructing Hardware in a Scala Embedded Language," [Online] Available: <https://chisel.eecs.berkeley.edu/>.

U. T. Ltd. (2015) Performance monitoring using ultrasoc. [Online].

U. T. Ltd. Soc Debug. [Online]. Available: V. Academy, "Introduction to uvm," [Online] Available: <https://verificationacademy.com/sessions/introduction-uvm>.

W. Ramirez, H. Gomez, and E. Roa, "On UVM Reliability in Mixed-Signal Verification," in 2019 IEEE 10th Latin American Symposium on Circuits Systems (LASCAS), Feb 2019, pp. 233–236.

W. Ramirez and E. Roa, "Post-Silicon Debugging Platform with Bus Monitoring Capability to Perform Behavioral and Performance Analyses," in 2019 IEEE 10th Latin American Symposium on Circuits Systems (LASCAS), Feb 2019, pp. 81–84.

Xilinx, "Artix-7 35T Arty FPGA Evaluation Kit." [Online]. Available: <https://www.xilinx.com/products/boards-and-kits/arty.html#documentation>

Y. N. Yun, J. B. Kim, N. D. Kim, and B. Min, "Beyond UVM for Practical soc Verification," in 2011 International SoC Design Conference, Nov 2011, pp. 158–162.