Alma Mater Studiorum – University of Bologna

Master of Science in Electronic Engineering

Field of

Digital Architectures for Data Processing

Debug Module for the NanoRV Processor

Supervisor:

Prof. Davide Rossi

Co- Supervisor:

Prof. Francesco Conti

Prof. Francisco Rodríguez Ballester

Candidate:

Lorenza Guerriero

A.A. 2025/2026

To myself, with sincere affection.

Index

1. Introduction

1.1 Context and Motivation

In recent years, the rise of open hardware initiatives have significantly transformed the landscape of processor design and embedded system development. Among these, the RISC-V instruction set architecture (ISA) has emerged as a prominent and influential standard, offering a modular and extensible framework free from licensing constraints. RISC-V's growing adoption across both academia and industry reflects its appeal for educational use, research purposes, and commercial implementations alike. In this context, the **'nanorv' processor**, a compact implementation of the RISC-V RV32I ISA, presents an ideal platform for experimentation and customization, especially in applications where simplicity, scalability, and open design are essential.

Debugging represents a fundamental aspect of processor development and embedded systems design. A robust debug infrastructure enables developers to inspect the internal state of the processor, set breakpoints, step through instructions, and detect faults during execution. Without appropriate debugging mechanisms, identifying the root causes of hardware or software errors becomes highly time-consuming and error-prone. This is particularly crucial in resource-constrained environments, where visibility into the system's behavior is inherently limited.

The motivation for this thesis stems from the need to provide **a fully functional debug module** tailored for the nanorv processor. Given the processor's minimalist architecture, integrating a debug solution that is both non-intrusive and compatible with standard debugging tools (e.g., those using the JTAG interface) is a non-trivial challenge. The design and implementation of such a module aim to bridge the gap between the lightweight nature of nanorv and the rich debugging capabilities required in real-world applications.

1.2 Objectives of the Thesis

The primary objective of this thesis is to **design, implement, and validate a debug module** for the nanorv processor, in compliance with the RISC-V Debug Specification

and compatible with widely adopted debugging standards such as JTAG. The module must offer essential debug functionalities—such as halting, resuming, and single-stepping the processor—while maintaining a clean integration with the existing nanorv architecture.

Specifically, this work sets out to:

- Analyze the debug requirements of RISC-V compliant processors, with a focus on minimal implementations like nanorv. In particular, the current state of the nanorv design doesn't support exceptions or interrupts, so the traditional approach to set a breakpoint (inserting a halt/debug instruction that'll trigger an exception when executed) can't be used;
- Design a hardware-based debug module that enables low-level interaction with the processor's state, including access to registers and memory;
- Define and implement the communication interfaces between the debug module, the processor core, and external debugging tools;
- Integrate the module with a JTAG interface, enabling external control through standard debugging protocols;
- Validate the functionality of the module through simulation and practical test environments.

Secondary objectives include providing detailed documentation, modular code for ease of reuse, and insights into the integration of the debug module with open-source development tools. These goals are aimed not only at addressing immediate debugging needs but also at offering a reusable blueprint for future extensions or applications involving nanorv or similar processors.

1.3 Structure of the Thesis

This thesis is structured into several chapters, each addressing a specific aspect of the work carried out. Following this introduction:

- **Chapter 2** provides a general overview of the RISC-V architecture, focusing on the RV32I subset, and introduces the nanorv processor. It also discusses the role and importance of debugging in the broader context of embedded systems.
- **Chapter 3** is dedicated to the **debug module** itself. It includes a detailed description of the module's architecture, functionality, and hardware interfaces. The chapter also elaborates on the communication mechanisms with the processor and the external world.
- **Chapter 4** explores debugging from a broader perspective, discussing common debugging techniques, available software tools, and the integration of the debug module with development environments.
- **Chapter 5** focuses on the **JTAG interface**, explaining its architecture and how it is used in debugging contexts. The chapter also outlines the steps required to configure and utilize JTAG in conjunction with the designed module.
- **Chapter 6** concludes the thesis by presenting a summary of the results, reflecting on the effectiveness of the implemented solution, and proposing potential directions for future improvements and developments.

Finally, the **appendices** contain relevant technical material, including the full source code of the debug module and additional documentation that supports replication or further development.

This structure aims to offer a clear and coherent narrative, guiding the reader through the conceptual, architectural, and practical aspects of implementing a debug module in a modern open-source processor environment.

2. Overview of the RISC-V System

The growing demand for open, flexible, and efficient processor architectures has positioned **RISC-V** as a pivotal standard in both academic and industrial domains. Unlike proprietary instruction set architectures, RISC-V offers a **free and extensible ISA** that fosters transparency, long-term stability, and architectural experimentation. This openness has enabled the emergence of lightweight cores for embedded systems, as well as high-performance multicore implementations.

This chapter introduces the foundational aspects of the RISC-V system architecture, with a focus on the **RV32I** base ISA used in this project. We explore the instruction set structure, encoding schemes, register organization, and the design philosophy that emphasizes **simplicity, modularity, and portability**. These characteristics make RISC-V particularly well-suited for system-on-chip (SoC) development and custom hardware designs—such as the one implemented and debugged in this thesis.

We also highlight the role of modular instruction set extensions, the versioning mechanism maintained by **RISC-V International**, and the advantages that this architecture brings to hardware/software co-design. Understanding the baseline architecture is critical, as it shapes the implementation decisions for both the processor core and the debugging infrastructure described in later chapters.

2.1 RV32I Architecture of the RISC-V Family

The RV32I base integer instruction set constitutes the foundational unprivileged ISA for RISC-V architectures. It comprises 40–47 instructions, depending on the implementation: arithmetic, logic, load/store, branch, and system calls (e.g., ECALL, EBREAK). The register file consists of 32 general-purpose 32-bit registers (x0 to x31), with x0 permanently hardwired to zero, supporting deterministic behavior and simplified hardware design.

RV32I uses fixed 32-bit instruction encoding, employing six instruction formats: R, I, S, B, U, and J. This enables a load-store architectural model, where all memory operations are performed via specific instructions, while arithmetic and logical

operations operate exclusively on registers. The spec ensures symmetric encoding patterns that facilitate pipelining and simple decoding logic.

Importantly, the ISA is modular: RV32I is supplemented by optional, standardized extensions (e.g., M for multiply/divide, A for atomic operations, C for compressed instructions) that can improve performance or reduce footprint, depending on application requirements. The specification is ratified by RISC-V International—the unprivileged ISA version 2.1 was ratified in May 2024, ensuring stability for toolchain support —. In designer notation, ISA variants are denoted as, for example, RV32IMC or RV32IMAFDC where multiple extensions are combined.

RV32I was specifically crafted to be minimal yet sufficient: it supports OS-level functionality and compiler targets while minimizing hardware complexity, making it ideal for embedded contexts and teaching environments. The spec allows implementations to treat SYSTEM instructions as traps or NOPs for a minimal core—this enables simplified implementations achieving even fewer than 40 hardware operations.

2.2 Main Features of the 'nanorv' Processor

The **nanorv** processor is the central processing unit of a custom System-on-Chip (SoC) developed for educational and experimental purposes. It is designed with minimalism and transparency in mind, emphasizing ease of understanding over raw performance. Its implementation is entirely written in VHDL, making it an ideal case study for students learning about digital systems, hardware description languages, and embedded processor design.

2.2.1 General Overview

At its core, nanorv is a **soft-IP CPU** designed to execute the **RISC-V RV32I** instruction set. It is a **multi-cycle, non-pipelined** processor that prioritizes simplicity over complexity. Unlike many modern processors, it does not implement advanced architectural features such as pipelining, cache memories, interrupts, or debug

interfaces. Instead, the design focuses on minimizing the use of logic elements, which significantly reduces FPGA resource usage.

The nanorv processor acts as a **bus master** on an **AMBA APB bus**, enabling it to initiate read and write transfers with various on-chip peripherals and memory blocks. Upon system reset, it begins fetching instructions from address 0x00000000, using an internal program counter (PC) to sequence execution.

2.2.2 CPU Architecture

The internal architecture of nanorv consists of several classic CPU components:

- **Program Counter (PC)**: Holds the address of the instruction to be fetched.
- **Instruction Register (IR)**: Stores the 32-bit instruction word retrieved from memory.
- **Register File**: Contains 32 general-purpose 32-bit registers (x0 to x31). Register x0 is hardwired to zero, as per the RISC-V specification.
- **Arithmetic Logic Unit (ALU)**: Executes arithmetic and logical operations using operands from the register file or instruction.
- **Memory Interface**: Implements the APB protocol to read/write data from memory and I/O.
- **Control Unit (CU)**: Decodes instructions and orchestrates the operations of the other components.

Each of these modules is implemented as a distinct VHDL component, reinforcing modularity and testability. All signals within the processor are synchronized with the rising edge of the system clock, except for the asynchronous, active-low reset signal.

2.2.3 Instruction Execution and Timing

The execution of an instruction in nanorv follows a **five-phase cycle** typical of von Neumann architectures:

1. **Fetch**: Read the instruction from memory using the PC.
2. **Decode**: Interpret the instruction to determine the operation and operands.
3. **Operand Fetch**: Retrieve operands from the register file.

4. **Execute**: Perform the operation via the ALU.
5. **Write-back**: Store the result and update the PC.

Due to its multi-cycle nature, the processor requires between **3 and 6 clock cycles** to execute a single instruction. This latency is partly due to the APB protocol, where each bus transfer requires a minimum of **2 clock cycles**, and some instructions (e.g., load/store) need multiple transfers. Unlike pipelined processors, nanorv does not execute multiple instructions simultaneously. This design decision simplifies debugging and signal tracing during simulation, which is especially beneficial for educational environments.

2.2.4 Instruction Set Support

Nanorv fully supports the **RV32I base instruction set**, which includes:

- Integer arithmetic (e.g., add, sub, addi)
- Logical operations (e.g., and, or, xor)
- Memory operations (lw, sw, lb, sh, etc.)
- Control flow instructions (beq, bne, jal, jalr)
- System instructions (ecall, ebreak), these instructions are treated as NOP (no-operation) instructions, as nanorv doesn't support priviledged instructions

The standard toolchain used to generate programs for the nanorv platform uses pseudo-instructions (e.g., li, mv) to enhance human readability, although they are translated into legal RV32I instructions by the assembler. These instructions can be interpreted using disassembler listings provided by the simulation tools. Register naming in nanorv follows RISC-V conventions, where registers are referred to by both their number (x0–x31) and mnemonic (zero, ra, sp, a0–a7, etc.).

2.2.5 Integration within the SoC

The nanorv CPU is embedded within a custom SoC architecture that includes:

- **Instruction and data memory blocks** (2 KB for code, 512 bytes for data)
- A simple **GPIO module** with read/write 32-bit ports

- A custom **multi-channel PWM peripheral**
- An **APB interconnect** with a slave decoder for address-based module selection

All components are instantiated in the apb_system_simple.vhd top-level entity, where the nanorv CPU plays the role of APB master. The slave decoder routes requests to the correct APB slave by analyzing CPU addresses and control signals. Figure 9 of the source document illustrates the entire SoC interconnection.

The CPU fetches instructions from the 0x00000000 address space and accesses data memory and peripherals using standard APB transfers. The simplicity of this setup makes the system predictable and easy to simulate or analyze, though it lacks flexibility for dynamic application development (e.g., it cannot download code dynamically or be debugged externally without extensions).

2.2.6 Development and Toolchain

The nanorv SoC workflow is based on a set of open-source and proprietary tools:

- **RISC-V GCC toolchain (riscv64-unknown-elf)** for compiling C and assembly programs
- **ihexconv**: a utility to convert compiled binaries into VHDL memory content files
- **Modelsim**: a VHDL simulator used to run the system in a testbench environment

The embedded application is developed separately, compiled to .elf format, then transformed into .hex and finally into a VHDL package (memory_contents_pkg.vhd) that initializes the instruction memory. This workflow, though automated, is relatively slow, as each change to the program requires recompiling the entire SoC design.

In the absence of debugging support, developers rely on waveform analysis (e.g., PC and IR signals) and simulation logs to validate processor behavior. This limitation reinforces the motivation for developing an integrated debug module, which would greatly improve test efficiency and usability.

2.3 Importance of Debugging in Embedded Systems

In embedded-system development, debugging plays a critical role for several reasons. Embedded devices often lack rich I/O interfaces (keyboards, displays, file systems), making traditional debugging approaches (console output, file logging) infeasible. As a result, **hardware-assisted debugging**—via interfaces like JTAG or specialized debug modules—is essential.

A robust debug infrastructure helps locate semantic bugs (e.g., incorrect logic) and low-level issues such as memory corruption or concurrency faults—areas known to consume significant debugging effort, especially in constrained environments. Studies show that memory and concurrency bugs are less frequent but disproportionately time-consuming, underlining the value of tools enabling precise inspection and control at runtime [1].

Embedded-software debugging is further challenged by real-time constraints, limited observability, and minimal on-chip resources. Research into embedded developer behavior shows that successful debugging often requires iterative fault localization and close inspection of program state, often using external trace or data retrieval tools for effective diagnosis [2]. Techniques such as in-circuit emulation (ICE) or on-chip debug support via standard interfaces like JTAG enable non-intrusive access to processor internals, even in production silicon.

Without integrated debug support, developers rely on intrusive methods like printf-style debugging or instrumented builds, which may alter timings and mask faults—a phenomenon especially problematic in embedded real-time systems.

Therefore, integrating a hardware debug module into nanorv (and similar architectures) equips developers with powerful capabilities: halting execution, single-stepping through instructions, setting breakpoints, and accessing registers and memory—all essential for reliable embedded-system development and validation.

3. Debug Module for RISC-V

Modern embedded processors require integrated debugging capabilities to support development workflows, test routines, and fault isolation. While high-end microcontrollers may offer complex trace units or vendor-specific debug subsystems, custom SoCs often lack such infrastructure, making debugging both intrusive and inefficient.

To address this challenge, a custom debug module was designed and integrated into the nanorv RISC-V SoC. The objective was to enable full control over the CPU during runtime via an external debugger, with functionality inspired by the Debug Module Interface (DMI) defined in the official RISC-V Debug Specification.

This hardware module implements a JTAG-accessible register interface that allows developers to halt execution, inspect and modify CPU state, read/write general-purpose registers, and access memory-mapped peripherals. The module was designed to be lightweight, synthesizable on entry-level FPGAs, and compatible with both simulation workflows and real hardware debugging via OpenOCD.

What follows in this chapter is a detailed analysis of its architecture, design decisions, and the VHDL implementation strategies adopted to ensure portability, observability, and robust integration with the rest of the SoC.

3.1 Description of the Debug Module

The debug_module implemented for the nanorv processor is a VHDL-based hardware component designed to enable external access to core internal data structures such as the **program counter**, **instruction register**, **register file**, and **memory interface**. It acts as a **bridge between the CPU and a JTAG interface**, allowing developers to monitor and control the processor's state in real-time, particularly during simulation or external debugging via hardware.

The core structure of the module is built around a **set of dedicated debug registers** (JTAG_USER_DATA_REG), which serve as a shared communication point between the CPU and the external interface. These include:

- **Control Register (JTAG_CTRL_REG)**: manages operational commands such as go/stop, reset, memory access, and register access.
- **Status Register (JTAG_STATUS_REG)**: reflects internal flags and states, including CPU activity, memory operations, and program counter usage.
- **Memory Address Register**: used to store the target address for memory read/write operations.
- **Write Data Register**: holds data to be written to memory.
- **Read Data Register**: returns data read from memory or the register file.

Follow a table with all fields of all JTAG register.

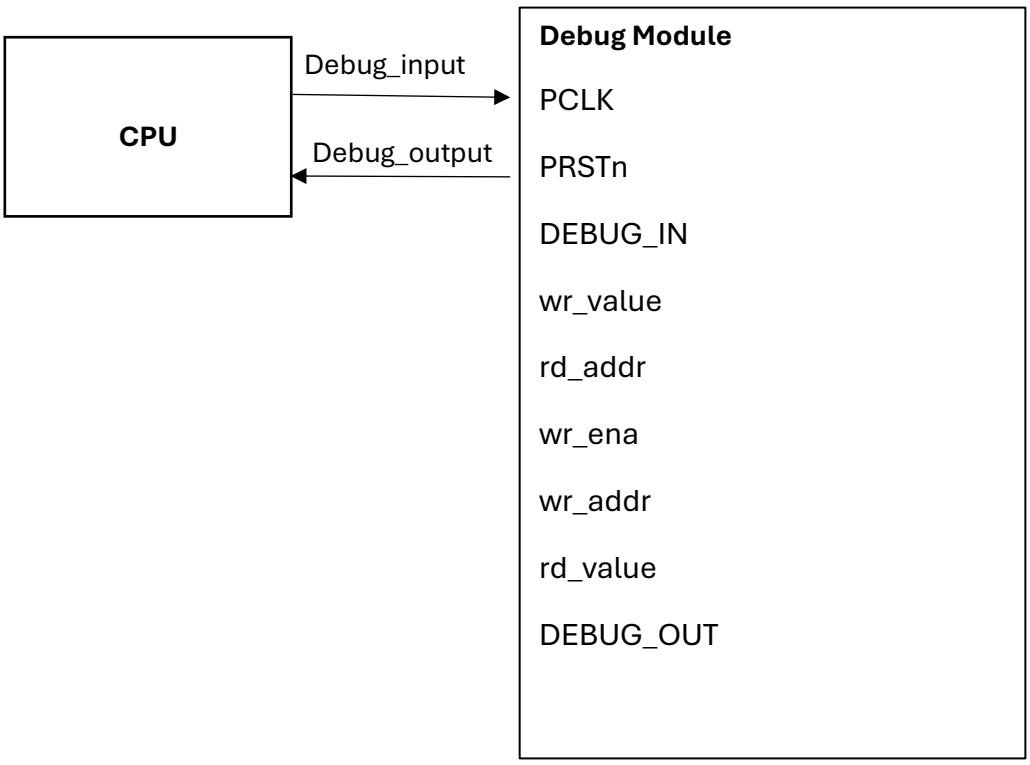
Register Name	Bits	Field Name	Description
Control Register JTAG_CTRL_REG	0	JTAG_CTRL_RSTN_BIT	Active-low reset control for the CPU.
	1	JTAG_CTRL_GO_BIT	Run/stop control (1 = Go, 0 = Stop).
	2	JTAG_CTRL_RDPC_BIT	Enable read of Program Counter.
	3	JTAG_CTRL_WRPC_BIT	Enable write to Program Counter.
	4:8	JTAG_CTRL_REGADDR	Target register address for register file access.
	9	JTAG_CTRL_RDREG_BIT	Enable read from register file.
	10	JTAG_CTRL_WRREG_BIT	Enable write to register file.
	11:12	JTAG_CTRL_MEMSIZE	Memory access size (00 = byte, 01 = halfword, 10 = word).
	13	JTAG_CTRL_RDMEM_BIT	Enable memory read.
	14	JTAG_CTRL_WRMEM_BIT	Enable memory write.

	15	JTAG_CTRL_RDMEM_UNS_BIT	Enable unsigned memory read.
Status Register JTAG_STATUS_REG	0	JTAG_STATUS_GO_BIT	Indicates if CPU is running (Go mode).
	1	JTAG_STATUS_MEM_BUSY_BIT	Memory interface is busy.
	2	JTAG_STATUS_PC_BUSY_BIT	Program Counter is being accessed.
	3	JTAG_STATUS_REG_BUSY_BIT	Register file interface is busy.
Memory Address Register JTAG_MEMADDR_REG	0:31	mem_addr	Address for memory read/write operations.
Write Data Register JTAG_WRDATA_REG	0:31	wr_data	Data to be written to memory or registers.
Read Data Register JTAG_RDDATA_REG	0:31	rd_data	Data read from memory or registers.

The module is architected to be **fully synchronous** with the system clock (PCLK) and supports **asynchronous reset** (PRSTn). It connects to the CPU through two custom record types: debug_inputs (providing the module with CPU state) and debug_outputs (used to control the CPU).

Through this interface, developers can halt the processor, inspect and modify register contents, and perform memory transactions in a controlled, non-intrusive manner. Internally, the module is governed by three distinct state machines that manage control flow, register file access, and memory transactions.

Following block diagram showing debug module and CPU interconnected with debug_inputs and debug_outputs. DEBUG_OUT and DEBUG_IN are records so include different signals to access at the program counter, register and memory (Debug_out: - rstn; - go; (to access PC) - pc_rd; - pc_wr; (to access registers) - reg_addr; - reg_rd; - reg_wr; (to access memory) - mem_size; - mem_addr; - mem_rd; - mem_wr; - read_unsigned; (shared by Pc, reg and mem); - wr_data) (Debug_in: - fetch; - mem_busy;-rd_data).



3.2 Functionality of the Module

The debug module offers a rich set of debugging functionalities specifically tailored for low-level embedded development. It provides:

- **External halting and resuming of CPU execution** via the jtag_go signal (bit in the control register).
- **Read and write access to the register file** through the reg_addr, reg_rd, and reg_wr signals.
- **Program counter (PC) inspection and overwrite**, controlled via read_pc and write_pc.
- **Memory read/write transactions**, handled through the mem_addr, mem_size, read_mem, write_mem, and read_unsigned fields.
- **CPU reset control**, using the rstn signal in the control register.

A key architectural decision was to decouple all functionality via a **register-mapped interface**, allowing software-controlled debugging without tight coupling to internal CPU implementation details. All operations are issued by writing to the control register and processed through synchronized state machines.

The **status register** acts as an important feedback mechanism. It provides flags such as:

- **GO**: Whether the CPU is currently running.
- **MEM_BUSY**, **REG_BUSY**, **PC_BUSY**: Indicators that memory, register file, or PC are currently being accessed or updated.

Furthermore, by combining control and status monitoring, the debug module supports **safe single-step execution**. This is achieved by toggling jtag_go and monitoring fetch from the CPU (received via DEBUG_IN.fetch) to ensure execution completes before halting again.

Data exchanges with the CPU are realized by writing or reading from JTAG_USER_DATA_REG. For example, during a memory read, the address is placed in the memory address register, a read is triggered via read_mem in the control register, and the result is later read from the rd_data register.

This functionality is especially useful in simulations where interactive debugging can be implemented by driving these control and data signals directly from a testbench.

3.3 Architecture of the Debug Module

Internally, the debug module defines a register file (JTAG_USER_DATA_REG) with five key registers:

- **Control Register** – used to send commands (e.g., GO, STOP, RESET, memory or register access).
- **Status Register** – provides flags indicating the current state (e.g., memory busy, core executing).
- **Memory Address Register** – indicates the memory location targeted for read/write.
- **Write Data Register** – contains data to be written to memory or registers.

- **Read Data Register** – holds the result of a read operation.

The internal architecture of the debug module is composed of three main subsystems, each managed by a dedicated **Finite State Machine (FSM)**:

1. **FSM_GO** – handles CPU execution state (GO, STOP, RESET).
2. **FSM_MEM** – manages memory operations.
3. **FSM_REG** – coordinates register file access.

These FSMs operate synchronously with the system clock (PCLK) and reset signal (PRSTn), maintaining consistent state transitions in the presence of external debug commands.

1. GO-State Controller (FSM_handle_go)

This FSM manages the CPU execution state:

- **GO**: Normal execution mode.
- **STOP**: CPU is halted; all activities suspended.
- **BUSY / FETCH_BUSY**: Transitional states for safe halting.
- **RESET**: Resets internal debug module logic.
- **P_STOP**: Pre-stop transitional state.

This logic uses the jtag_go bit and CPU fetch signal to coordinate controlled halting and resuming, ensuring instructions complete safely before halting.

2. Memory Controller (FSM_memory_operation)

Manages memory operations via:

- **IDLE**: Awaiting operation.
- **MEM_BUSY**: Active memory transaction.
- **DONE**: Operation completed, awaiting next command.

Signals involved: read_mem, write_mem, mem_addr, mem_size, wr_data, rd_data.

The jtag_mem_busy signal flags when the memory interface is in use and updates the corresponding bit in the status register.

3. Register File Controller (FSM_reg_file)

Handles access to the CPU's register file using:

- **REG_IDLE**: Idle state.
- **REG_BUSY**: A read/write operation is in progress.
- **REG_DONE**: Transaction complete.

Signals involved: reg_addr, reg_rd, reg_wr, and wr_data. A reg_busy_flag signal indicates activity and updates the status register accordingly.

4. JTAG Register File (JTAG_USER_DATA_REG)

This array of registers serves as the central configuration and data interface for the module. Index-based aliases provide access to:

- mem_addr, wr_data, rd_data
- status, control
- Register-level fields (e.g., reg_addr, read_pc, write_pc, read_mem, etc.)

Each field is accessed by reading from or writing to a specific index in the array, which mimics a **memory-mapped I/O interface** often seen in JTAG systems.

5. Data Path Logic

The jtag_wr process handles register writes, enforces write-protection for the first two read-only registers, and ensures that CPU read data (DEBUG_IN.rd_data) is saved into rd_data for external inspection.

The jtag_rd process simply returns the requested data from JTAG_USER_DATA_REG to the external world.

Subsystem	Description
GO FSM	Manages CPU execution (GO, STOP, etc.)
Memory FSM	Handles read/write operations to memory

Register FSM	Handles access to CPU register file
Status register	Tracks GO mode, memory, and PC activity
Control register	Issues commands: halt, reset, PC access
JTAG registers	Unified register map for control & data flow

3.4 Interfaces and Connections

The **debug module** developed for the *nanorv* processor acts as a bridge between the CPU and the external JTAG interface, enabling external debugging actions such as halting execution, inspecting and modifying registers, and reading or writing to memory. Its interfaces are carefully designed to ensure minimal intrusion in the CPU pipeline while offering robust control and visibility.

3.4.1 Involved Hardware Components

The main hardware components involved in the debug infrastructure are:

- **The nanorv_cpu processor** – a RISC-V RV32I implementation that exposes a dedicated debug interface via record ports (debug_inputs and debug_outputs). The CPU itself does not directly read or write the debug module's internal registers; instead, it provides signals representing its internal state (e.g., program counter, fetched instruction, register file outputs) and accepts external control signals (e.g., go, pause, read/write) from the debug module.
- **The JTAG interface** – used for external communication. In simulation, this is typically modeled as a set of processes that read/write the debug module's register file. In hardware, a physical JTAG adapter connects to the FPGA and drives the same register interface through a virtual JTAG core.
- **The APB-based SoC architecture** – the SoC in which the CPU and debug module are instantiated uses an APB bus for peripheral interconnection. However, the debug_module is not connected to the APB as a standard slave; instead, it is directly linked between the CPU's debug ports and the JTAG/adapter interface, operating outside the APB address map.

- **The debug_module entity** – implements the core debugging functionality and acts as a bridge between the CPU and the external debug interface (JTAG or simulation adapter). It maintains an internal register file for control, status, memory access, and register file access, which can be manipulated externally to observe or modify CPU state.

3.4.2 Communication with the Processor

The interaction between the debug module and the processor is realized through two custom VHDL records: debug_inputs and debug_outputs. These serve as structured buses allowing clean bidirectional communication:

- **debug_outputs (from CPU to debug module)** includes:
  - rd_data: result of read operations (PC, memory, or registers).
  - fetch: indicates whether the CPU is fetching an instruction.
  - mem_busy: indicates if a memory operation is ongoing.
- **debug_inputs (from debug module to CPU)** includes:
  - rstn: CPU reset signal (active low).
  - go: signal to allow or halt execution.
  - pc_rd, pc_wr: flags to read or write the program counter.
  - reg_addr, reg_rd, reg_wr: register file access control.
  - mem_addr, mem_rd, mem_wr, mem_size, read_unsigned: memory access parameters.
  - wr_data: data to be written (used by memory or register writes).

When debugging is active (go = '0'), the CPU routes memory, PC, and register operations through these debug interfaces, allowing external tools to intervene safely and predictably.

3.5 Communication Protocol with the Outside

The debug module communicates with the outside world primarily via the **JTAG interface**. In simulation environments, this connection is simulated using direct port mapping (e.g., JTAG_wr_value, JTAG_rd_addr), while in hardware setups, the system uses an **Altera Virtual JTAG** module paired with a **JTAG Debug Adapter**.

External tools interact with the module through a series of addressable registers:

- Write operations are performed via the wr_ena, wr_addr, and wr_value signals.
- Read operations are enabled by setting rd_addr, with the result provided on rd_value.

This communication protocol enables precise, low-level control over the CPU's behavior and state, including:

- Halting the CPU (GO = 0)
- Reading the program counter or registers
- Reading/writing memory at any address
- Injecting instructions or data

All of this is done without interrupting the processor’s operation, ensuring the debug process is both non-invasive and fully deterministic.

4. Debugging and Debug Tools

Debugging is not a peripheral activity in hardware design—it is an intrinsic part of the engineering process. As digital systems become more complex and heterogeneous, with tightly coupled processors, peripherals, and memory hierarchies, the ability to observe, control, and interpret internal behavior becomes both a necessity and a challenge.

In FPGA-based SoCs, traditional software debugging methods fall short due to limited visibility into internal signals and the absence of an operating system to host standard tools. Instead, dedicated hardware debugging infrastructures must be designed into the system from the ground up. These include mechanisms for halting the CPU, inspecting internal registers, and injecting commands from external tools—all without disturbing the system's functional correctness.

This chapter explores both the theoretical foundations and the practical implementation of debug features in the context of a RISC-V SoC. It provides insight into the debugging strategies adopted, the software tools employed during development, and the low-level architectural choices that enable full-system observability and control, whether during simulation or on real FPGA hardware.

4.1 Debugging Techniques

Embedded systems designers commonly adopt two complementary debugging paradigms: **run-stop (halt-based) debugging** and **real-time trace debugging**, as outlined in authoritative surveys of industrial SoCs [3][4].

- **Run-stop debugging**: Involves halting program execution at defined points—e.g., breakpoints—to inspect CPU state, memory, and control flow. This method ensures precise control but is intrusive, interrupting real-time behavior, which may alter fault conditions [5].
- **Real-time trace debugging**: Captures internal execution behavior asynchronously, allowing post-mortem analysis of program flow and memory access without stopping

the processor. Embedded trace cores, like ARM CoreSight or RISC-V proposals, implement this non-intrusive approach [6].

Another critical concept is **Design-for-Debug (DfD)**, which advocates embedding debug infrastructure within the SoC architecture from the outset to enable effective fault localization without requiring costly redesigns at later stages.

The on-chip debugging module developed in this thesis implements a **run-stop** approach via a JTAG-like register interface, enabling deterministic control of program flow, read/write access to registers and memory, and safe halt/resume execution — fully consistent with established DfD principles.

Importantly, the current debug module design does not preclude the adoption of **real-time trace debugging techniques** in future iterations. For example, since the module already allows examination of the program counter (PC) while the processor is running, a trace-enhanced version could continuously capture and stream PC values to reconstruct the execution flow, thereby enabling low-intrusion program tracing alongside the run-stop capabilities.

4.2 Software Tools for Debugging

In this project, we utilized several industry-standard software tools customized for the RISC-V-based nanorv SoC:

- **ModelSim**: Employed for VHDL simulation of the nanorv CPU and the debug module. ModelSim provides signal waveforms (e.g., PC, IR, APB bus lines) and allows interactive control of debug register transactions for simulation of external JTAG commands.
- **Intel Quartus Prime**: Used for FPGA synthesis, particularly targeting the Cyclone family with integration of the altera_mf MegaFunction library. Quartus enables both functional and timing simulation, and inclusion of debug-support flags (DEBUG_SUPPORT := true) to instantiate the debug module.
- **Doxygen**: Utilized to generate structured documentation from the commented VHDL source (using --! \brief annotations). It produces HTML and PDF outputs describing

entities, registers, alias mappings, FSM states, and interface specifications, greatly enhancing code maintainability and clarity.

- **OpenOCD (Open On-Chip Debugger)**: Used for real hardware debugging once the SoC is deployed on FPGA. OpenOCD facilitates JTAG-based control using standard RISC-V DMI commands (dmi_read, dmi_write, progbuf, sysbus, abstract) to access debug modules and memory via GDB [7].

  To support OpenOCD on Windows, we installed **nodist** (Node.js distribution) and **GnuWin32** (Unix toolset for Windows), allowing execution of Tcl scripts and cross-platform utilities for target communication.

  This combination provides a seamless debugging workflow both in simulation and on real hardware, fully leveraging open-source RISC-V tooling.

4.3 Integration of the Debug Module with Development Tools

The integration of the debug_module into the development toolchain involves orchestrated coordination between simulation, synthesis, documentation, and runtime tools.

1. **ModelSim-based testbench**: The debug module is instantiated alongside the CPU in a custom testbench. Control signals (wr_ena, wr_addr, rd_addr) are driven via Tcl or VHDL stimulus, simulating interaction with a host. Outputs (rd_value, status, DEBUG_OUT) are observed via waveform. This method enables interactive validation of register map behavior and FSM transitions.
2. **Quartus synthesis for FPGA deployment**: When configured with DEBUG_SUPPORT = true, the SoC includes the debug_module. In simulation mode, the adapter is bypassed via direct port mapping; in hardware mode (Cyclone FPGA), the altera_virtual_jtag and jtag_adapter components convert physical JTAG signals into internal register transactions. This enables external tools to communicate with the module.
3. **Documentation with Doxygen**: The annotated VHDL code is processed to generate browsable documentation, including descriptions of:

a. Control and status registers (addresses, bit mapping)
b. FSM states in memory and register controllers
c. Alias names used (e.g. mem_addr, wr_data, status)
d. Interface records (debug_inputs, debug_outputs)

   This facilitates easier onboarding and maintenance for future developers or extensions.

4. **OpenOCD + GDB hardware debugging**: In hardware mode, OpenOCD interfaces with the FPGA via JTAG. Using RISC-V DMI commands, GDB can access the debug module's register file (JTAG_USER_DATA_REG), control execution flow (GO/STOP), and perform read/write operations on memory and registers. This fully supports run-stop debugging, breakpoints, and memory inspection without modifying the CPU design—leveraging the debug module's register-mapped protocol.

   This integrated toolchain (ModelSim, Quartus, Doxygen, OpenOCD/GDB) provides a coherent workflow for:

- Designing and verifying the debug module in simulation,
- Documenting and maintaining the module,
- Executing hardware-assisted debugging on physical FPGA boards.

5. JTAG Interface

In complex digital systems, especially in processor-based or SoC designs, observing and controlling internal states during execution is a fundamental aspect of debugging. Traditional simulation environments provide some visibility, but they become impractical when moving to physical hardware. In this context, the JTAG interface offers a standardized and low-pin-count method to access internal signals, memories, and registers, even after synthesis and implementation.

In this project, JTAG is not only a physical interface but also a conceptual bridge between external debugging tools and the internal debug logic. While hardware vendors provide official support for JTAG-based debugging, implementing a custom communication channel using the virtual JTAG features of Intel FPGAs enables greater control and flexibility. This choice influences both the architecture and simulation approach of the design, requiring careful integration with custom VHDL modules.

The following sections explore the technical details of the JTAG protocol, its structural components, and the rationale behind its use in this work, both in simulation and real hardware scenarios.

5.1 Introduction to JTAG

JTAG (Joint Test Action Group) is a standard (IEEE 1149.1) developed to facilitate testing and debugging of integrated circuits (ICs) and complex systems on chip (SoCs). Originally intended for boundary scan testing, JTAG has become a fundamental tool in modern digital design, especially for tasks like in-system programming, verification, and on-chip debugging.

The core idea behind JTAG is to provide access to the internal registers and logic of an integrated circuit via a standardized serial interface. This allows developers to observe and control the internal state of a system without requiring physical access to every internal node — which would be impractical or impossible in deeply embedded systems.

JTAG interfaces are now commonly found in FPGAs, microcontrollers, processors, and custom logic blocks. The protocol is characterized by a small number of dedicated pins: Test Clock (TCK), Test Mode Select (TMS), Test Data In (TDI), Test Data Out (TDO), and an optional reset (TRST). Through a state machine known as the TAP (Test Access Port), it becomes possible to shift data into and out of internal scan chains and instruction registers.

In this project, the JTAG interface is used not only for boundary scan, but primarily to implement a **debug communication channel** between a host (e.g., OpenOCD on a PC) and the internal debug module inside the FPGA.

5.2 Architecture and Functioning of JTAG

The architecture of a typical JTAG implementation revolves around the **TAP controller**, a finite state machine that manages access to two main registers:

- **Instruction Register (IR):** selects which operation or module the JTAG interface will interact with.
- **Data Register (DR):** used to shift in data (e.g., values to write) and shift out data (e.g., values to read).

The TAP controller follows a fixed state diagram dictated by the IEEE standard, including states such as Test-Logic-Reset, Run-Test/Idle, Shift-IR, Shift-DR, Capture-DR, and Update-DR.

In our design, the core JTAG interaction is abstracted through the use of **Altera's altera_virtual_jtag component**, which acts as a JTAG interface inside the FPGA, allowing us to expose virtual instruction and data registers accessible by tools like OpenOCD. This component handles the TAP FSM internally and provides clean signals to the rest of the system, such as shift_dr, capture_dr, update_dr, and the serial data lines tdi and tdo.

To interpret and manage these signals, a **jtag_debug_adapter** module is implemented. This VHDL component acts as a bridge between the JTAG TAP interface and the

internal debug logic, decoding commands, writing to debug registers, and returning read values via a scan chain. It effectively converts the low-level serial protocol into parallel control signals usable by the system's debug logic.

This adapter defines its own protocol format over the Data Register (DR), typically structured to include fields such as write enable, write address, write value, and read address. On receiving a full JTAG scan operation (shifted in and updated), the adapter extracts the command and drives the corresponding operations internally.

5.3 Configuration of the JTAG Interface

In order to use the JTAG interface in the project, a layered configuration was developed. At the hardware level, the altera_virtual_jtag IP component from Intel Quartus was instantiated. This module automatically exposes the physical JTAG interface of the FPGA and allows creation of one or more virtual JTAG instances, each with its own instruction and data registers.

The configuration includes:

- Instruction register width and decoding.
- Number of virtual JTAG instances.
- Interface signals (tck, tdi, tdo, shift_dr, update_dr, capture_dr, etc.) connected to the custom debug logic.

The JTAG interface is then connected to a custom jtag_debug_adapter module. Inside this adapter, a shift register (or scan chain) is defined to process incoming data from the JTAG host. This register is 64 bits wide in the current implementation, enough to support complex transactions (e.g., read and write combined).

Furthermore, a generic parameter SIMULATION_MODE was introduced. This allows the same design to support two modes of operation:

- **Simulation mode**: Bypasses the virtual JTAG and allows low-level testbenches to directly stimulate the debug interface by writing to the shift register signals in simulation.

- **Hardware mode**: Enables full JTAG interaction via the physical JTAG interface and a real adapter (e.g., OpenOCD connected to an FTDI or USB-Blaster).

This dual-mode design greatly improves testability and flexibility, making it possible to validate the design both in simulation and on real hardware with the same core logic.

5.4 Applications and Use of JTAG in Debugging

The most powerful use of JTAG in this project is for debugging purposes. By integrating the altera_virtual_jtag interface and the custom debug adapter, the system gains the ability to inspect and control its internal state at runtime without invasive modifications to the logic.

Key debugging functionalities enabled by this JTAG setup include:

- **Read and write access to debug registers**: Enables real-time manipulation of CPU internals, including general-purpose registers, program counters, and control flags.
- **Memory access**: Allows direct interaction with memory-mapped regions, useful for analyzing variables, peripheral registers, or injecting test patterns.
- **Instruction and program loading**: Small test routines or instructions can be loaded dynamically into memory via JTAG.
- **Integration with OpenOCD**: Open-source tools like OpenOCD are supported to interface with the debug system using standard RISC-V debug commands (dmi_read, dmi_write, etc.), making debugging sessions scriptable and automatable.

The altera_virtual_jtag interface is used in **hardware mode**, enabling high-speed, non-intrusive communication with internal logic when the system is deployed on an FPGA. In **simulation mode**, the JTAG protocol is emulated by directly driving the shift registers and control signals, enabling thorough pre-silicon validation.

This dual-path approach draws inspiration from and is compatible with the architecture proposed in the Advanced Debug System [8], an open-source reference project implementing modular debug over JTAG for soft processors. The design philosophy is

aligned with this project's goals: modularity, compatibility with open tooling, and robustness across development and production environments.

6. Methodology

The development of the debug module for the *nanorv* processor followed an **incremental and iterative methodology**, combining theoretical study, code analysis, and systematic simulation.

The first phase consisted of an extensive review of the available documentation on the *nanorv* SoC and its RV32I CPU. This study was fundamental to understand the baseline architecture, the interaction between the CPU and its peripherals, and the feasibility of extending the design with dedicated debugging capabilities.

Subsequently, the VHDL source code of the processor and the SoC was examined in detail. Through this analysis, the structural hierarchy, signal interconnections, and functional processes were identified. This step enabled the definition of a clear integration point for the debug module without altering the core datapath of the CPU.

The next step involved the use of the already available simulation environment. In fact, the initial testbench was provided together with the SoC and was fully functional, although it did not include any reference to the debug module. In the early stages, ModelSim and the testbench were mainly used to observe the overall behavior of the system and to better understand the SoC operation, without introducing structural modifications. This allowed starting from a stable and reliable simulation baseline, which later served as a reference for the subsequent extensions.

Once the use of the simulation framework was consolidated, the design of the debug interfaces — represented by the *debug_inputs* and *debug_outputs* records — was initiated. These records define the structured communication channel between the CPU and the debug module. Their incremental introduction made it possible to expose essential internal CPU information — such as the program counter value, the current instruction, and the register contents — while at the same time providing the processor with external control signals (halt, resume, single-step).

The practical implementation proceeded in stages. First, the **halt/resume mechanism** was implemented, ensuring that the CPU could be paused whenever the go signal was

deasserted and resumed when it was asserted again. After validating this functionality, the focus shifted to **read operations**, enabling external access to CPU state, registers, and memory. Only after the read operations were reliably functioning, **write operations** were introduced, allowing controlled modification of internal CPU state and memory content.

The final phase of the methodology focused on consolidation and refinement. The registers were structured into a coherent register file accessible via the debug module, the VHDL code was enriched with consistent comments and Doxygen-compatible documentation, and a **procedure file** was developed for the testbench. This file replaced repetitive signal-driving code with reusable procedures, greatly improving the readability and maintainability of the simulation environment.

7. Practical Work

The practical implementation of the debug module began with the definition and integration of the **debug I/O records**. The *debug_inputs* record was used to capture the CPU status signals, such as the *fetch* signal (to check the progress of instruction execution), the *mem_busy* signal (to monitor whether the memory is busy or free), and the *reg_data* signal (to read data from the register file). Conversely, the *debug_outputs* record provided the control signals to the CPU, enabling read and write access to the program counter, the register file, and memory. This interface ensured a clear separation between the processor core and the external debug infrastructure.

The next step was the incremental design of the module's internal **state machines**. Initially, a simple controller was implemented to verify that the CPU could be halted and resumed deterministically via the go signal. This step was essential to validate the core principle of run-stop debugging. Once this was achieved, the register file access logic was introduced. The design supported read requests first, as they are non-intrusive: the CPU's internal state could be observed without modifying its behavior. Only in a subsequent stage were write operations introduced, enabling modifications of registers and memory.

Each new feature was introduced in isolation and verified through simulation. For instance, before enabling write access to memory, the read functionality was thoroughly tested with different addresses and data sizes. This staged approach minimized the risk of introducing hard-to-trace errors and made the debugging process itself more manageable.

To support usability and maintainability, the VHDL source was enriched with detailed comments and structured documentation. Using Doxygen, a documentation set was generated that automatically described entities, ports, registers, and processes.

Finally, a **library of procedures** was developed to automate common tasks within the testbench. Instead of manually coding long sequences of signal toggling for each simulation, the procedures encapsulated operations such as *"write to register," "read*

from memory," or "halt CPU". This significantly reduced testbench verbosity, transforming what could have been thousands of repetitive code lines into compact, readable simulation scripts.

8. Results and Analysis

The experimental validation demonstrated that the debug module was successfully integrated into the *nanorv* SoC and operated as intended.

The **halt/resume mechanism** worked reliably: whenever the go signal was deasserted, the CPU execution paused deterministically, and it resumed seamlessly when the signal was asserted again. This provided precise control over program execution, confirming the run-stop principle of the design.

The **read functionality** was verified across the program counter, register file, and memory. External tools could issue a read request together with an address in one clock cycle, and the corresponding value was made available on the subsequent cycle. This deterministic one-cycle latency proved to be consistent across all simulations.

The **write functionality** was likewise confirmed. By issuing a write command along with the target address and data, the CPU's register file and memory were successfully updated, with the operation completed in a predictable cycle-by-cycle manner.

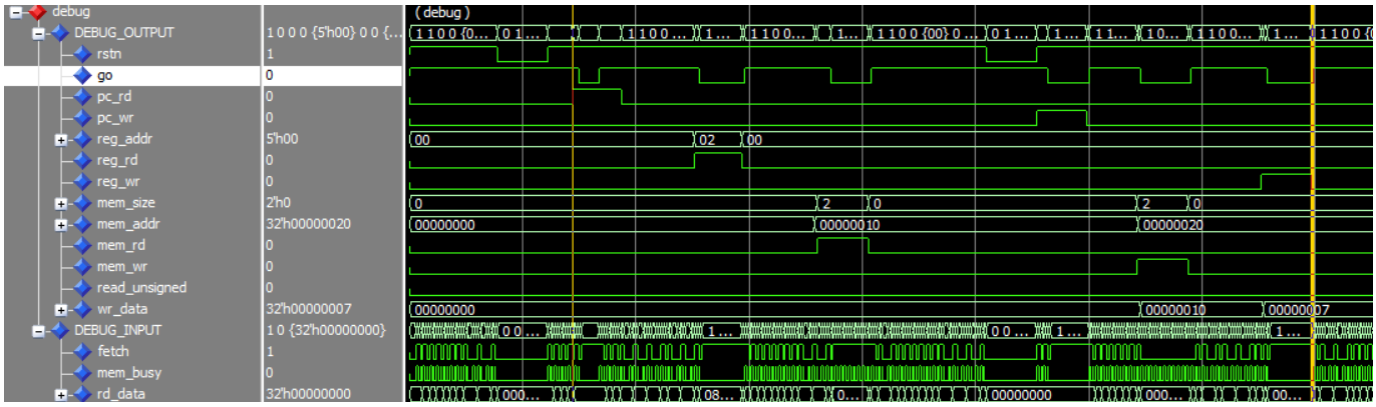
From a usability standpoint, the introduction of a structured register file and documented interface significantly improved accessibility. The Doxygen-generated documentation provided an immediate reference for developers, and the testbench procedure library drastically simplified simulation workflows.

In summary, the results show that:

- The CPU can be halted and resumed on demand.
- Register and memory reads execute with a **1-cycle request and 1-cycle response latency**.
- Register and memory writes complete correctly and predictably.
- The debug module enables deterministic and reproducible control of the CPU without introducing instability.

These results validate both the functional correctness of the debug module and its integration methodology, demonstrating that the module can serve as a robust foundation for extended debug features, including potential trace-based enhancements.

The picture shows that the different reading and writing of the debug module.



9. Conclusions

This thesis presented the design, implementation, and validation of a hardware debug module for the *nanorv* processor, addressing the need for controllability and observability in embedded systems. The work demonstrated how a structured, incremental approach—spanning from theoretical analysis to practical implementation and simulation—can produce a robust component that improves the usability and reliability of a custom RISC-V core. The results confirm that debugging is not merely an auxiliary feature, but a fundamental element of processor design, enabling efficient development, verification, and future extensibility.

9.1 Reflections on the Results Obtained

The work presented in this thesis achieved the objective of designing and integrating a functional debug module for the *nanorv* processor. The module successfully provides deterministic run–stop control, read/write access to program counter, registers and memory, and a clean interface with the CPU through dedicated debug records. Simulations confirmed that the CPU can be halted and resumed on demand, and that both read and write operations are executed with predictable cycle-level accuracy.

The incremental methodology adopted—starting from documentation study, through progressive implementation of halt/resume, reads, and writes, and culminating in documentation and testbench automation—proved highly effective. It minimized design errors, facilitated verification, and ensured a smooth integration process. Furthermore, the generation of structured documentation and reusable testbench procedures enhanced the maintainability and accessibility of the project.

9.2 Possible Future Developments

While the implemented debug module fulfills the core requirements, several avenues remain open for future improvement:

- **Trace-based debugging**: as discussed in the methodology, the current design could be extended with real-time trace capabilities. By continuously sampling the program

counter and optionally data accesses, the module could reconstruct program execution flow without halting the CPU, enabling more advanced performance analysis and fault detection.

- **Integration with higher-level tools**: connecting the debug module to established frameworks such as OpenOCD or GDB would allow seamless source-level debugging on top of the hardware mechanisms.
- **Enhanced testbench automation**: while the current procedure library improves readability, more sophisticated verification environments (e.g., SystemVerilog UVM or cocotb) could be explored to further strengthen validation.
- **FPGA prototyping and hardware validation**: extending beyond simulation, the debug module could be synthesized on an FPGA platform (such as Intel Cyclone) and connected to physical JTAG interfaces to assess timing, performance, and usability in a real hardware environment.

9.3 Final Considerations

This thesis demonstrated the feasibility and effectiveness of embedding a structured debug module within the *nanorv* processor. By adhering to **Design-for-Debug principles**, the module not only enables controlled observation and manipulation of CPU state but also lays the groundwork for more advanced debugging methodologies.

From a broader perspective, the project illustrates how academic-scale processor designs can benefit from professional-level debugging infrastructures, bridging the gap between didactic systems and industrial-grade SoCs. The developed module provides a solid foundation for both research and education, contributing to the advancement of open-source RISC-V platforms and their ecosystem.

10. Acknowledgements

I would like to thank my supervisor, Professor Davide Rossi, for allowing me to undertake this work, most of which was conducted at the UPV in Valencia, Spain. There, I met Professor Francisco Rodríguez Ballester, a truly genuine and helpful person whom I sincerely thank. Finally, I would also like to thank my co-supervisor, Francesco Conti, for his continuous support. I thank all the faculty members I met during this master's program for allowing me to be here now. A special thought also goes to my entire family, my fiancée, to those who are here and to those who are no longer, for always giving me support and affection, in both the good times and the difficult ones.

11. Bibliography

[1] ‘What we can learn from how programmers debug their code’ by Thomas Hirsch, Birgit Hofer. https://doi.org/10.48550/arXiv.2103.12447

[2] ‘Debugging Behaviour of Embedded-Software Developers: An Exploratory Study’ by Pansy Arafa, Daniel Solomon, Samaneh Navabpour, Sebastian Fischmeister. https://doi.org/10.48550/arXiv.1704.03397

[3] ‘Functional Debug Techniques for Embedded Systems’ by Bart Vermeulen https://doi.org/10.1109/MDT.2008.66

[4] ‘OpenOCD configurations for RISC-V’. https://eclipse-embed-cdt.github.io/debug/openocd/riscv/

[5] ‘Real-time trace : A Better Way to Debug Embedded Applications’ by James Campbell, Valeriy Kazantsev, Hugh O’Keeffe. https://api.semanticscholar.org/CorpusID:199517100

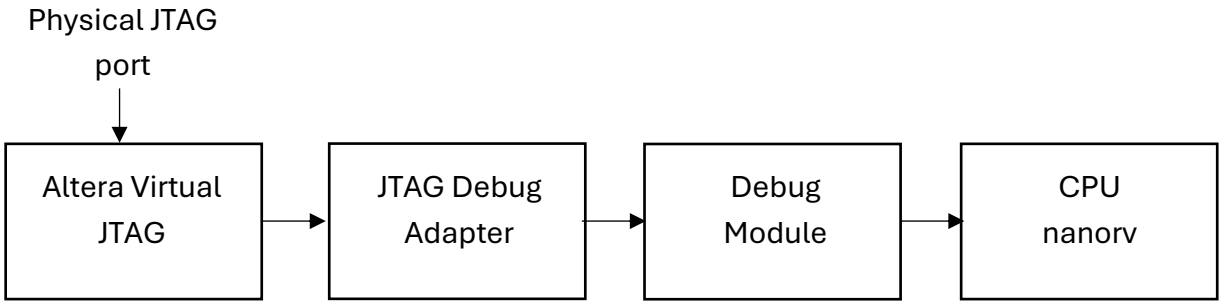
[6] A. Weiss et al., ‘Understanding and Fixing Complex Faults in Embedded Cyberphysical Systems’, in Computer, vol. 54, no. 1, pp. 49-60, Jan. 2021, doi: 10.1109/MC.2020.3029975.

[7] ‘Development of On-Chip Debugging Module for RISC-V Processor’ by Sergey Kornev, Vladimir Andreev. https://www.researchgate.net/publication/378284780_Development_of_On-Chip_Debugging_Module_for_RISC-V_Processor

[8] Advanced Debug System by FreeCores. https://github.com/freecores/adv_debug_sys

12. Appendix

The **Physical JTAG Port** provides the external connection between the host debugger and the FPGA device through the standard JTAG interface (TCK, TDI, TDO, TMS). Inside the FPGA, the **Altera Virtual JTAG** IP core bridges this physical port to the user logic, exposing JTAG control signals to custom hardware modules. The **JTAG Debug Adapter** translates the low-level JTAG scan operations into read/write commands for the internal registers of the debug system. These commands are handled by the **Debug Module**, which manages CPU control (start, stop, reset) as well as memory and register access. Finally, the **CPU core** executes the instructions and responds to the debug commands, enabling full external debugging through the JTAG infrastructure.



12.1 Source Code of the Debug Module

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.nanorv_pkg.all;
entity debug_module is
  port (
    PCLK    : in std_logic;
    PRSTn   : in std_logic;
    DEBUG_IN  : in debug_inputs;
```

```
      wr_value  : in std_logic_vector(31 downto 0);
      rd_addr   : in std_logic_vector(JTAG_REG_ADDR_WIDTH-1 downto 0);
      wr_ena    : in std_logic;
      wr_addr   : in std_logic_vector(JTAG_REG_ADDR_WIDTH-1 downto 0);
      rd_value  : out std_logic_vector(31 downto 0);
      DEBUG_OUT : out debug_outputs ;
end entity debug_module;
use work.nanorv_pkg.all;
architecture behavioral of debug_module is
   signal actual_go : std_logic;
   signal jtag_mem_busy : std_logic;
   type jtag_regfile_t is array(0 to JTAG_REG_NUMBER-1) of std_logic_vector (31 downto
0);
   signal JTAG_USER_DATA_REG : jtag_regfile_t;
   alias mem_addr  : std_logic_vector(31 downto 0) is
JTAG_USER_DATA_REG(JTAG_MEMADDR_REG;
   alias wr_data   : std_logic_vector(31 downto 0) is
JTAG_USER_DATA_REG(JTAG_WRDATA_REG);
   alias rd_data   : std_logic_vector(31 downto 0) is
JTAG_USER_DATA_REG(JTAG_RDDATA_REG);
   alias status    : std_logic_vector(31 downto 0) is
JTAG_USER_DATA_REG(JTAG_STATUS_REG);
 --! JTAG control register, used to control the state of the module and to read and write
the values of the program counter, the instruction register and the memory interface.
The JTAG control register has the following bits:
   alias rstn     : std_logic is
JTAG_USER_DATA_REG(JTAG_CTRL_REG)(JTAG_CTRL_RSTN_BIT);
   alias jtag_go   : std_logic is
JTAG_USER_DATA_REG(JTAG_CTRL_REG)(JTAG_CTRL_GO_BIT);
   alias read_pc   : std_logic is
JTAG_USER_DATA_REG(JTAG_CTRL_REG)(JTAG_CTRL_RDPC_BIT);
```

```
  alias write_pc  : std_logic is
JTAG_USER_DATA_REG(JTAG_CTRL_REG)(JTAG_CTRL_WRPC_BIT);

  alias reg_addr  : std_logic_vector(4 downto 0) is
JTAG_USER_DATA_REG(JTAG_CTRL_REG)(JTAG_CTRL_REGADDR_BIT_H downto
JTAG_CTRL_REGADDR_BIT_L);

  alias reg_rd    : std_logic is
JTAG_USER_DATA_REG(JTAG_CTRL_REG)(JTAG_CTRL_RDREG_BIT);

  alias reg_wr    : std_logic is
JTAG_USER_DATA_REG(JTAG_CTRL_REG)(JTAG_CTRL_WRREG_BIT);

  alias mem_size  : std_logic_vector(1 downto 0) is
JTAG_USER_DATA_REG(JTAG_CTRL_REG)(JTAG_CTRL_MEMSIZE_BIT_H downto
JTAG_CTRL_MEMSIZE_BIT_L);

  alias read_mem  : std_logic is
JTAG_USER_DATA_REG(JTAG_CTRL_REG)(JTAG_CTRL_RDMEM_BIT);

  alias write_mem : std_logic is JTAG_USER_DATA_REG
(JTAG_CTRL_REG)(JTAG_CTRL_WRMEM_BIT);

  alias read_unsigned : std_logic is
JTAG_USER_DATA_REG(JTAG_CTRL_REG)(JTAG_CTRL_RDMEM_UNS_BIT);

  --! The FSMs are used to handle the state of the module during the operation with the
memory and the register file of the CPU.

  type state_GO is (GO, STOP, BUSY, FETCH_BUSY, RESET, P_STOP);

  signal go_current_state, go_next_state : state_GO;

  type state_MEM is (IDLE, MEM_BUSY, DONE);

  signal mem_current_state, mem_next_state : state_MEM;

  type state_REG is (REG_IDLE, REG_BUSY, REG_DONE);

  signal reg_current_state, reg_next_state : state_REG;

  signal reg_busy_flag : std_logic;

begin

  -- CPU Connection

  DEBUG_OUT.rstn     <= rstn;

  DEBUG_OUT.go       <= actual_go;
```

```
  DEBUG_OUT.pc_rd    <= read_pc;
  DEBUG_OUT.pc_wr    <= write_pc;
  DEBUG_OUT.reg_addr  <= reg_addr;
  DEBUG_OUT.reg_rd   <= reg_rd;
  DEBUG_OUT.reg_wr   <= reg_wr;
  DEBUG_OUT.mem_size  <= mem_size;
  DEBUG_OUT.mem_addr  <= mem_addr;
  DEBUG_OUT.mem_rd   <= read_mem;
  DEBUG_OUT.mem_wr   <= write_mem;
  DEBUG_OUT.read_unsigned <= read_unsigned;
  DEBUG_OUT.wr_data   <= wr_data;
  FSM_go_update:process(PCLK,PRSTn) is --! The process is used to update the state of the module (GO, RESET, BUSY, FETCH_BUSY, STOP and P_STOP)
  begin
    if (PRSTn = '0') then
      go_current_state <= RESET;
    elsif rising_edge(PCLK) then
      go_current_state <= go_next_state;
    end if;
  end process FSM_go_update;
  FSM_handle_go: process (go_current_state, jtag_go, DEBUG_IN.fetch) is --! The process is used to handle the state of the debug module depending on the fetch of the CPU, jtag_go signal and the current state of the module
  begin
    case go_current_state is
      when GO =>
        if (jtag_go = '0' and DEBUG_IN.fetch = '1') then
          go_next_state <= FETCH_BUSY;
        elsif (jtag_go = '0' and DEBUG_IN.fetch = '0') then
```

```
      go_next_state <= BUSY;
    end if;
    if (DEBUG_OUT.rstn = '0') then
      go_next_state <= RESET;
    end if;
    actual_go <= '1';
  when STOP =>
    if (jtag_go = '1') then
      go_next_state <= GO;
    end if;
    if (DEBUG_OUT.rstn = '0') then
      go_next_state <= RESET;
    end if;
    actual_go <= '0';
  when BUSY =>
    if (DEBUG_IN.fetch = '1') then
      go_next_state <= P_STOP;
    end if;
    if (DEBUG_OUT.rstn = '0') then
      go_next_state <= RESET;
    end if;
    actual_go <= '1';
  when FETCH_BUSY =>
    if (DEBUG_IN.fetch = '0') then
      go_next_state <= BUSY;
    end if;
    if (DEBUG_OUT.rstn = '0') then
      go_next_state <= RESET;
```

```
            end if;
            actual_go <= '1';
          when RESET =>
            if (DEBUG_OUT.rstn = '1') then
              go_next_state <= GO;
            end if;
            actual_go <= '1';
          when P_STOP =>
            if (DEBUG_OUT.rstn = '1') then
              go_next_state <= STOP;
            else
              go_next_state <= RESET;
            end if;
            actual_go <= '1';
        end case;
    end process FSM_handle_go;
    FSM_memory_update: process(PCLK,PRSTn) is --! The process is used to update the state of the memory module (IDLE, MEM_BUSY and DONE)
    begin
      if (PRSTn = '0') then
        mem_current_state <= IDLE;
      elsif rising_edge(PCLK) then
        mem_current_state <= mem_next_state;
      end if;
    end process FSM_memory_update;
   FSM_memory_operation: process (mem_current_state, read_mem, write_mem, actual_go) is --! The process is used to handle the state of the debug module during the operation with the memory of the CPU
    begin
```

```
      case mem_current_state is
        when IDLE =>
          if (read_mem = '1' or write_mem = '1') then
            mem_next_state <= MEM_BUSY;
          else
            mem_next_state <= IDLE;
          end if;
          jtag_mem_busy <= '0';
        when MEM_BUSY =>
          if (actual_go = '0') then
            mem_next_state <= DONE;
          else
            mem_next_state <= MEM_BUSY;
          end if;
          jtag_mem_busy <= '1';
        when DONE => -- wait for writing on to the control register, with jtag_mem_busy <=
'0';
          if (read_mem = '0' and write_mem = '0') then
            mem_next_state <= IDLE;
          else
            mem_next_state <= DONE;
          end if;
          jtag_mem_busy <= '0';
      end case;
    end process FSM_memory_operation;
    -- FSM Register File
    FSM_reg_file_update: process(PCLK,PRSTn) is --! The process is used to update the
state of the register file (REG_IDLE, REG_BUSY and REG_DONE)
      begin
```

```
    if (PRSTn = '0') then
      reg_current_state <= REG_IDLE;
    elsif rising_edge(PCLK) then
      reg_current_state <= reg_next_state;
    end if;
  end process FSM_reg_file_update;
```

FSM_reg_file: process(reg_current_state, reg_wr, reg_rd, actual_go) is --! The process is used to handle the state of the debug module during the operation with the register file of the CPU

```
  begin
    case reg_current_state is
    when REG_IDLE =>
      if (reg_wr = '1' or reg_rd = '1') then
        reg_next_state <= REG_BUSY;
      else
        reg_next_state <= REG_IDLE;
      end if;
      reg_busy_flag <= '0';
    when REG_BUSY =>
      if (actual_go = '0') then
        reg_next_state <= REG_DONE;
      else
        reg_next_state <= REG_BUSY;
      end if;
      reg_busy_flag <= '1';
    when REG_DONE =>
      if (reg_rd = '0' and reg_wr = '0') then
        reg_next_state <= REG_IDLE;
      else
```

```
            reg_next_state <= REG_DONE;
          end if;
          reg_busy_flag <= '0';
        end case;
      end process FSM_reg_file;
  jtag_wr: process(PCLK, PRSTn) is
  --! The JTAG connection is used to connect the CPU to the JTAG interface. It uses the JTAG_USER_DATA_REG register file to write or read the values from JTAG to the instruction register, the program counter and the memory interface of the CPU.
  --! Update JTAG status register every clock cycle, if a reset event comes the register are settet to zero. Remember that the first two registers are only reading.
  begin
    if (PRSTn = '0') then
      JTAG_USER_DATA_REG <= (others => (others => '0'));
      JTAG_USER_DATA_REG (JTAG_CTRL_REG)(JTAG_CTRL_RSTN_BIT) <= '1';
      JTAG_USER_DATA_REG (JTAG_CTRL_REG)(JTAG_CTRL_GO_BIT) <= '1';
      status <= (others => '0');
      rd_data <= (others => '0');
    elsif rising_edge(PCLK) then
      -- Update JTAG status register every clock cycle
      status (JTAG_STATUS_GO_BIT)      <= actual_go;
      status (JTAG_STATUS_MEM_BUSY_BIT) <= jtag_mem_busy;
      status (JTAG_STATUS_REG_BUSY_BIT) <= reg_busy_flag;
      if (read_pc = '1' or write_pc = '1' ) then
        status(JTAG_STATUS_PC_BUSY_BIT) <= '1';
      else
        status(JTAG_STATUS_PC_BUSY_BIT) <= '0';
      end if;
      rd_data <= DEBUG_IN.rd_data;
```

```
        if ((wr_ena = '1') and (to_integer(unsigned(wr_addr))>= 2)) then
          JTAG_USER_DATA_REG(to_integer(unsigned(wr_addr))) <= wr_value;
        end if;
      end if;
  end process jtag_wr;
  jtag_rd: process(PCLK) is --! \brief JTAG Read process
  begin
      if rising_edge(PCLK) then
        rd_value <= JTAG_USER_DATA_REG(to_integer(unsigned(rd_addr)));
      end if;
  end process jtag_rd;
end architecture behavioral;
```

12.2 Procedures Used

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.nanorv_pkg.all;
package procedure_pkg is
   --Register constants
   constant JTAG_CTRL_GO      : std_logic_vector(31 downto 0) := (JTAG_CTRL_RSTN_BIT
=> '1', JTAG_CTRL_GO_BIT => '1', others => '0');
   constant JTAG_CTRL_STOP    : std_logic_vector(31 downto 0) :=
(JTAG_CTRL_RSTN_BIT => '1', JTAG_CTRL_GO_BIT => '0', others => '0');
   constant JTAG_CTRL_GORST   : std_logic_vector(31 downto 0) :=
(JTAG_CTRL_RSTN_BIT => '0', JTAG_CTRL_GO_BIT => '1', others => '0');
   constant JTAG_CTRL_RST     : std_logic_vector(31 downto 0) := (JTAG_CTRL_RSTN_BIT
=> '0', JTAG_CTRL_GO_BIT => '0', others => '0');
```

```
  constant JTAG_CTRL_RDPC_STOP : std_logic_vector(31 downto 0) :=
(JTAG_CTRL_RSTN_BIT => '1', JTAG_CTRL_GO_BIT => '0', JTAG_CTRL_RDPC_BIT => '1',
others => '0');

  constant JTAG_CTRL_RDPC_GO   : std_logic_vector(31 downto 0) :=
(JTAG_CTRL_RSTN_BIT => '1', JTAG_CTRL_GO_BIT => '1', JTAG_CTRL_RDPC_BIT => '1',
others => '0');

  constant JTAG_CTRL_WRPC     : std_logic_vector(31 downto 0) :=
(JTAG_CTRL_RSTN_BIT => '1', JTAG_CTRL_GO_BIT => '0', JTAG_CTRL_WRPC_BIT => '1',
others => '0');

  constant mem_size           : std_logic_vector(1 downto 0) := "10"; -- 10 = 32 bits, 01 =
16 bits, 00 = 8 bits, 11 = 64 bits ???

  constant JTAG_CTRL_RDMEM_STOP   : std_logic_vector(31 downto 0) :=
(JTAG_CTRL_RSTN_BIT => '1', JTAG_CTRL_GO_BIT => '0', JTAG_CTRL_RDMEM_BIT => '1',
JTAG_CTRL_MEMSIZE_BIT_H downto JTAG_CTRL_MEMSIZE_BIT_L => mem_size, others
=> '0');

  constant JTAG_CTRL_WRMEM_STOP   : std_logic_vector(31 downto 0) :=
(JTAG_CTRL_RSTN_BIT => '1', JTAG_CTRL_GO_BIT => '0', JTAG_CTRL_WRMEM_BIT => '1',
JTAG_CTRL_MEMSIZE_BIT_H downto JTAG_CTRL_MEMSIZE_BIT_L => mem_size, others
=> '0');

  constant addr_reg0       : std_logic_vector (4 downto 0) := "00000";

  constant addr_reg1       : std_logic_vector (4 downto 0) := "00010";

  constant JTAG_CTRL_RDREG_ADDR : std_logic_vector(31 downto 0) :=
(JTAG_CTRL_RSTN_BIT => '1', JTAG_CTRL_GO_BIT => '0', JTAG_CTRL_RDREG_BIT => '1',
JTAG_CTRL_REGADDR_BIT_H downto JTAG_CTRL_REGADDR_BIT_L => addr_reg1,
others => '0');

  constant JTAG_CTRL_WRREG_ADDR : std_logic_vector(31 downto 0) :=
(JTAG_CTRL_RSTN_BIT => '1', JTAG_CTRL_GO_BIT => '0', JTAG_CTRL_WRREG_BIT => '1',
JTAG_CTRL_REGADDR_BIT_H downto JTAG_CTRL_REGADDR_BIT_L => addr_reg0,
others => '0');

  signal reading_value : std_logic_vector(31 downto 0) := (others => '0');

  -- JTAG signals

  signal JTAG_wr_ena   : std_logic := '0';

  signal JTAG_wr_addr  : std_logic_vector(JTAG_REG_ADDR_WIDTH-1 downto 0) :=
(others => '0');
```

```
  signal JTAG_wr_value : std_logic_vector(31 downto 0) := (others => '0');
  signal JTAG_rd_addr  : std_logic_vector(JTAG_REG_ADDR_WIDTH-1 downto 0) := (others => '0');
  signal JTAG_rd_value : std_logic_vector(31 downto 0) := (others => '0');
    --FSM state type debugging
  type tb_type is (W_RST_OFF, W_GO1, GO1, W_RST1, RST1, W_GO2, GO2, W_RST2, RST2, W_RST3, RST3, W_RST4, RST4,
            reading_pc, stop_reading_pc, reading_mem, stop_reading_mem, reading_regfile, stop_reading_regfile,
            writing_pc, stop_writing_pc, writing_mem, stop_writing_mem, writing_regfile, stop_writing_regfile);
  signal tb_state : tb_type := W_RST_OFF;
  -- Procedure declarations
  procedure wait_clk       (signal clk: std_logic; num: integer);
  procedure find_clk       (signal clk, find: std_logic; value: std_logic);
  procedure jtag_write      (signal JTAG_wr_ena: out std_logic;
                  signal JTAG_wr_addr : out std_logic_vector (JTAG_REG_ADDR_WIDTH-1 downto 0);
                  signal JTAG_wr_value : out std_logic_vector(31 downto 0);
                  signal clk: std_logic;
                  reg: in integer;
                  value: std_logic_vector(31 downto 0));
  procedure jtag_write2reg   (signal JTAG_wr_ena: out std_logic;
                  signal JTAG_wr_addr : out std_logic_vector (JTAG_REG_ADDR_WIDTH-1 downto 0);
                  signal JTAG_wr_value : out std_logic_vector(31 downto 0);
                  signal clk: std_logic;
                  value1: std_logic_vector(31 downto 0);
                  value2: std_logic_vector (31 downto 0));
  procedure jtag_rdmem      (signal clk : std_logic;
```

```
                signal value: out std_logic_vector(31 downto 0);
                signal JTAG_rd_addr: out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);
                signal JTAG_rd_value: in std_logic_vector (31 downto 0));
  procedure jtag_control_stop(signal JTAG_wr_ena: out std_logic;
                signal JTAG_wr_addr : out std_logic_vector (JTAG_REG_ADDR_WIDTH-1
downto 0);
                signal JTAG_wr_value : out std_logic_vector(31 downto 0);
                signal JTAG_rd_addr  : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);
                signal clk: std_logic);
  procedure cpu_rst_go      (signal tb_state     : out tb_type;
                signal JTAG_wr_ena    : out std_logic;
                signal JTAG_wr_addr   : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);
                signal JTAG_wr_value  : out std_logic_vector(31 downto 0);
                signal CLOCK_50      : in  std_logic;
                signal PRSTn         : in  std_logic);
  procedure read_pc         (signal JTAG_wr_ena   : out std_logic;
                signal tb_state      : out tb_type;
                signal JTAG_wr_addr   : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);
                signal JTAG_wr_value  : out std_logic_vector(31 downto 0);
                signal CLOCK_50      : in  std_logic);
  procedure read_mem        (signal JTAG_wr_ena   : out std_logic;
                signal tb_state      : out tb_type;
                signal JTAG_wr_addr   : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);
                signal JTAG_wr_value  : out std_logic_vector(31 downto 0);
```

```
                signal JTAG_rd_addr   : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);
                signal CLOCK_50     : in  std_logic;
                signal reading_value  : out std_logic_vector(31 downto 0);
                signal JTAG_rd_value  : in  std_logic_vector(31 downto 0);
                signal address      : in std_logic_vector (31 downto 0));
  procedure read_regfile    (signal JTAG_wr_ena   : out std_logic;
                signal tb_state     : out tb_type;
                signal JTAG_wr_addr   : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);
                signal JTAG_wr_value  : out std_logic_vector(31 downto 0);
                signal JTAG_rd_addr   : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);
                signal CLOCK_50     : in  std_logic);
  procedure write_pc       (signal JTAG_wr_ena  : out std_logic;
                signal tb_state     : out tb_type;
                signal JTAG_wr_addr  : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);
                signal JTAG_wr_value : out std_logic_vector(31 downto 0);
                signal CLOCK_50     : in  std_logic;
                constant pc_value   : in  std_logic_vector(31 downto 0));
    procedure write_mem       (signal JTAG_wr_ena   : out std_logic;
                signal tb_state     : out tb_type;
                signal JTAG_wr_addr   : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);
                signal JTAG_wr_value  : out std_logic_vector(31 downto 0);
                signal JTAG_rd_addr   : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);
                signal CLOCK_50     : in  std_logic;
                signal mem_value    : in std_logic_vector(31 downto 0);
```

```
                signal address      : in std_logic_vector (31 downto 0));
  procedure write_regfile    (signal JTAG_wr_ena   : out std_logic;
                signal tb_state     : out tb_type;
                signal JTAG_wr_addr   : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);
                signal JTAG_wr_value  : out std_logic_vector(31 downto 0);
                signal JTAG_rd_addr   : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);
                signal CLOCK_50     : in  std_logic;
                signal reg_value    : in std_logic_vector(31 downto 0));
end package procedure_pkg;
package body procedure_pkg is
  procedure wait_clk(signal clk: std_logic; num: integer) is
  begin
    for i in 0 to num-1 loop
      wait until rising_edge(clk);
    end loop;
    wait for 5 ns;
  end procedure wait_clk;
 procedure find_clk(signal clk, find: std_logic; value: std_logic) is
  begin
    while true loop
      wait until rising_edge(clk);
      wait for 5 ns;
      if (find = value) then
        exit;
      end if;
    end loop;
  end procedure find_clk;
```

```
  -- JTAG write register procedure
  procedure jtag_write(signal JTAG_wr_ena: out std_logic;
                       signal JTAG_wr_addr : out std_logic_vector (JTAG_REG_ADDR_WIDTH-1
downto 0);
                       signal JTAG_wr_value : out std_logic_vector(31 downto 0);
                       signal clk: std_logic;
                       reg: in integer;
                       value: std_logic_vector(31 downto 0)) is
  begin
    JTAG_wr_addr <= std_logic_vector(to_unsigned(reg, JTAG_wr_addr'length));
    JTAG_wr_value <= value;
    JTAG_wr_ena <= '1';
    wait_clk(clk, 1);
    JTAG_wr_ena <= '0';
  end procedure jtag_write;
  -- JTAG write CONTROL register and MEMORY ADDRESS register procedure
  procedure jtag_write2reg(signal JTAG_wr_ena: out std_logic;
                       signal JTAG_wr_addr : out std_logic_vector (JTAG_REG_ADDR_WIDTH-1
downto 0);
                       signal JTAG_wr_value : out std_logic_vector(31 downto 0);
                       signal clk: std_logic;
                       value1: std_logic_vector(31 downto 0);
                       value2: std_logic_vector (31 downto 0)) is
  begin
    JTAG_wr_ena <= '1';
    JTAG_wr_addr <= std_logic_vector(to_unsigned(JTAG_CTRL_REG,
JTAG_wr_addr'length));
    JTAG_wr_value <= value1;
    wait_clk(clk, 1);
```

```
    JTAG_wr_addr <= std_logic_vector(to_unsigned(JTAG_MEMADDR_REG,
JTAG_wr_addr'length));

    JTAG_wr_value <= value2;

    wait_clk(clk, 1);

    JTAG_wr_ena <= '0';

  end procedure jtag_write2reg;

  -- JTAG read procedure

  procedure jtag_rdmem   (signal clk : std_logic;

                signal value: out std_logic_vector(31 downto 0);

                signal JTAG_rd_addr: out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);

                signal JTAG_rd_value: in std_logic_vector (31 downto 0)) is

  begin

    JTAG_rd_addr <= std_logic_vector(to_unsigned(JTAG_MEMADDR_REG,
JTAG_rd_addr'length));

    wait_clk(clk, 1);

    value <= JTAG_rd_value;

    wait_clk(clk, 1);

  end procedure jtag_rdmem;

  -- JTAG stop CPU procedure for reading registers

  -- This procedure pausing the execution *and* reads the status register waiting for the
actual_go to be '0'

  procedure jtag_control_stop(signal JTAG_wr_ena: out std_logic;

                signal JTAG_wr_addr : out std_logic_vector (JTAG_REG_ADDR_WIDTH-1
downto 0);

                signal JTAG_wr_value : out std_logic_vector(31 downto 0);

                signal JTAG_rd_addr  : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);

                signal clk: std_logic) is

  begin
```

```
    -- Set rd address to get the value of the status register (actual_go is bit 0)
    JTAG_rd_addr <= std_logic_vector(to_unsigned(JTAG_STATUS_REG,
JTAG_rd_addr'length));
    JTAG_wr_addr <= std_logic_vector(to_unsigned(JTAG_CTRL_REG,
JTAG_wr_addr'length));
    JTAG_wr_value <= JTAG_CTRL_STOP;
    JTAG_wr_ena <= '1';
    wait_clk(clk, 1);
    JTAG_wr_ena <= '0';
    while (JTAG_rd_value(0) = '1') loop
      wait_clk(clk, 1);
    end loop;
  end procedure jtag_control_stop;
  -- JTAG reset and go the CPU
  -- This procedure wait the reset is disactivated, then the CPU goes, then do reset
operation and then CPU starts again
  procedure cpu_rst_go (  signal tb_state      : out tb_type;
                signal JTAG_wr_ena    : out std_logic;
                signal JTAG_wr_addr   : out std_logic_vector (JTAG_REG_ADDR_WIDTH-1
downto 0);
                signal JTAG_wr_value  : out std_logic_vector (31 downto 0);
                signal CLOCK_50      : in  std_logic;
                signal PRSTn         : in  std_logic) is
  begin
    tb_state <= W_RST_OFF;
    wait until (PRSTn = '1');
    tb_state <= W_GO1;
    wait_clk(CLOCK_50, 10);
    -- CPU is going --
```

```
    jtag_write(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, CLOCK_50,
JTAG_CTRL_REG, JTAG_CTRL_GO); -- go = 1, rstn = 1

    tb_state <= GO1;

    wait_clk(CLOCK_50, 15);

    tb_state <= W_RST1;

    -- Reset CPU after 25 clock cycles --

    wait_clk(CLOCK_50, 10);

    jtag_write(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, CLOCK_50,
JTAG_CTRL_REG, JTAG_CTRL_RST); -- go = 0, rstn = 0

    tb_state <= RST1;

    wait_clk(CLOCK_50, 10);

    tb_state <= W_GO2;

    jtag_write(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, CLOCK_50,
JTAG_CTRL_REG, JTAG_CTRL_GORST); -- go = 1, rstn = 0

    wait_clk(CLOCK_50, 10);

    -- CPU restart from the beginning executing normally --

    jtag_write(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, CLOCK_50,
JTAG_CTRL_REG, JTAG_CTRL_GO); -- go = 1, rstn = 1

    tb_state <= GO2;

    wait_clk(CLOCK_50, 10);

  end procedure cpu_rst_go;

  -- This procedure read the PC after 10 clk cycles when the go is 0 and after other 10 clk
cycles when the go is 1

  procedure read_pc ( signal JTAG_wr_ena   : out std_logic;

            signal tb_state     : out tb_type;

            signal JTAG_wr_addr  : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);

            signal JTAG_wr_value  : out std_logic_vector(31 downto 0);

            signal CLOCK_50     : in  std_logic) is

  begin
```

```
    tb_state <= reading_pc;
    -- Read PC when GO = 0 (CPU stopped)
    jtag_write(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, CLOCK_50,
JTAG_CTRL_REG, JTAG_CTRL_RDPC_STOP);
    wait_clk(CLOCK_50, 10);
     -- Read PC when GO = 1 (CPU running)
    jtag_write(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, CLOCK_50,
JTAG_CTRL_REG, JTAG_CTRL_RDPC_GO);
    wait_clk(CLOCK_50, 10);
    tb_state <= stop_reading_pc;
    jtag_write(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, CLOCK_50,
JTAG_CTRL_REG, JTAG_CTRL_GO); -- go = 1, rstn = 1
    wait_clk(CLOCK_50, 10);
  end procedure read_pc;
  procedure write_pc (signal JTAG_wr_ena   : out std_logic;
            signal tb_state     : out tb_type;
            signal JTAG_wr_addr  : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);
            signal JTAG_wr_value : out std_logic_vector(31 downto 0);
            signal CLOCK_50     : in  std_logic;
            constant pc_value    : in  std_logic_vector(31 downto 0)) is
  begin
    tb_state <= writing_pc;
    -- The CPU is stopped, write new PC value
    jtag_write(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, CLOCK_50,
JTAG_CTRL_REG, JTAG_CTRL_WRPC);
    jtag_write (JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, CLOCK_50,
JTAG_WRDATA_REG, pc_value);
    wait_clk(CLOCK_50, 20);
    -- Restart the CPU (GO = 1)
```

```
    jtag_write(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, CLOCK_50,
JTAG_CTRL_REG, JTAG_CTRL_GO);

    tb_state <= stop_writing_pc;

    wait_clk(CLOCK_50, 10);

  end procedure write_pc;

  -- This procedure ensure that the CPU is stopped and then read the memory address
specified in JTAG_MEMADDR_REG, the CPU still stopped

  procedure read_mem (signal JTAG_wr_ena    : out std_logic;

                      signal tb_state       : out tb_type;

                      signal JTAG_wr_addr   : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);

                      signal JTAG_wr_value  : out std_logic_vector(31 downto 0);

                      signal JTAG_rd_addr   : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);

                      signal CLOCK_50       : in  std_logic;

                      signal reading_value  : out std_logic_vector(31 downto 0);

                      signal JTAG_rd_value  : in  std_logic_vector(31 downto 0);

                      signal address        : in std_logic_vector (31 downto 0)) is

  begin

    wait_clk(CLOCK_50, 10);

    tb_state <= reading_mem;

    -- Pause CPU and wait for actual_go = 0

    jtag_control_stop(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, JTAG_rd_addr,
CLOCK_50); -- go= 0, rstn = 1

    -- Set memory address

    jtag_write(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, CLOCK_50,
JTAG_MEMADDR_REG, address);

    -- Issue memory read request

    jtag_write(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, CLOCK_50,
JTAG_CTRL_REG, JTAG_CTRL_RDMEM_STOP); -- go = 0, rstn = 1, read_mem = 1
```

```
    -- Perform actual memory read
    jtag_rdmem(CLOCK_50, reading_value, JTAG_rd_addr, JTAG_rd_value);
    tb_state <= stop_reading_mem;
    wait_clk(CLOCK_50, 20);
    -- Restore CPU to GO state
    jtag_write(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, CLOCK_50,
JTAG_CTRL_REG, JTAG_CTRL_GO); -- go = 1, rstn = 1
    wait_clk(CLOCK_50, 10);
  end procedure read_mem;
procedure write_mem (signal JTAG_wr_ena    : out std_logic;
            signal tb_state      : out tb_type;
            signal JTAG_wr_addr   : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);
            signal JTAG_wr_value  : out std_logic_vector(31 downto 0);
            signal JTAG_rd_addr   : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);
            signal CLOCK_50      : in  std_logic;
            signal mem_value     : in std_logic_vector(31 downto 0);
            signal address       : in std_logic_vector (31 downto 0)) is
  begin
    tb_state <= writing_mem;
    -- Stop the CPU (go = 0)
    jtag_control_stop(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, JTAG_rd_addr,
CLOCK_50);
    -- Command to write memory
    jtag_write(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, CLOCK_50,
JTAG_CTRL_REG, JTAG_CTRL_WRMEM_STOP);
    -- Set memory address to write
    jtag_write(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, CLOCK_50,
JTAG_MEMADDR_REG, address);
```

```
    -- Set the value to write in memory
    jtag_write(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, CLOCK_50,
JTAG_WRDATA_REG, mem_value);
    wait_clk(CLOCK_50, 20);
    -- CPU is going (go = 1)
    jtag_write(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, CLOCK_50,
JTAG_CTRL_REG, JTAG_CTRL_GO);
    tb_state <= stop_writing_mem;
    wait_clk(CLOCK_50, 10);
  end procedure write_mem;
  -- This procedure ensure that the CPU is stopped and then read the register address
specified in JTAG_CTRL_REGADDR, the CPU still stopped
  procedure read_regfile (signal JTAG_wr_ena    : out std_logic;
                signal tb_state      : out tb_type;
                signal JTAG_wr_addr   : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);
                signal JTAG_wr_value  : out std_logic_vector(31 downto 0);
                signal JTAG_rd_addr   : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);
                signal CLOCK_50      : in  std_logic) is
  begin
    -- Wait 10 clock cycles
    tb_state <= reading_regfile;
    wait_clk(CLOCK_50, 10);
    -- Pause CPU and wait for actual_go = 0
    jtag_control_stop(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, JTAG_rd_addr,
CLOCK_50); -- go= 0, rstn = 1
    -- Set register address and trigger the register to be read
    jtag_write(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, CLOCK_50,
JTAG_CTRL_REG, JTAG_CTRL_RDREG_ADDR);
```

```
    wait_clk(CLOCK_50, 20);
    tb_state <= stop_reading_regfile;
    -- Restore CPU to GO state
    jtag_write(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, CLOCK_50,
JTAG_CTRL_REG, JTAG_CTRL_GO); -- go = 1, rstn = 1
    wait_clk(CLOCK_50, 10);
  end procedure read_regfile;
  procedure write_regfile (signal JTAG_wr_ena    : out std_logic;
                signal tb_state      : out tb_type;
                signal JTAG_wr_addr   : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);
                signal JTAG_wr_value  : out std_logic_vector(31 downto 0);
                signal JTAG_rd_addr   : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1
downto 0);
                signal CLOCK_50      : in  std_logic;
                signal reg_value     : in std_logic_vector(31 downto 0)) is
  begin
    tb_state <= writing_regfile;
    wait_clk(CLOCK_50, 10);
    -- Stop the CPU
    jtag_control_stop(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, JTAG_rd_addr,
CLOCK_50);  -- go = 0, rstn = 1
    -- Set the register address to write and enable the writing
    jtag_write(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, CLOCK_50,
JTAG_CTRL_REG, JTAG_CTRL_WRREG_ADDR);
    -- Set the register value to write
    jtag_write(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, CLOCK_50,
JTAG_WRDATA_REG, reg_value);
    wait_clk(CLOCK_50, 20);
    -- CPU is going (go = 1)
```

```
    jtag_write(JTAG_wr_ena, JTAG_wr_addr, JTAG_wr_value, CLOCK_50,
JTAG_CTRL_REG, JTAG_CTRL_GO);
    tb_state <= stop_writing_regfile;
    wait_clk(CLOCK_50, 10);
  end procedure write_regfile;
end package body procedure_pkg;
```

12.3 Jtag Adapter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.nanorv_pkg.all;
entity jtag_debug_adapter is
  port (
    PCLK            : in  std_logic;
    PRSTn           : in  std_logic;
    -- Signals from altera_virtual_jtag
    jtag_tck        : in  std_logic;
    jtag_tdi        : in  std_logic;
    jtag_shift_dr   : in  std_logic;
    jtag_capture_dr : in  std_logic;
    jtag_update_dr  : in  std_logic;
    jtag_test_logic_reset: in  std_logic;
    jtag_run_test_idle  : in  std_logic;
    -- Signals to altera_virtual_jtag
    jtag_tdo        : out std_logic;
    -- Signals to debug_module (output of this adapter)
    debug_wr_value  : out std_logic_vector(31 downto 0);
```

```
    debug_rd_addr      : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1 downto 0);
    debug_wr_ena       : out std_logic;
    debug_wr_addr      : out std_logic_vector(JTAG_REG_ADDR_WIDTH-1 downto 0);
    -- Signals from debug_module (input to this adapter)
    debug_rd_value     : in  std_logic_vector(31 downto 0)
  );
end entity jtag_debug_adapter;
architecture behavioral of jtag_debug_adapter is
  -- Define the scan chain length of the Data Register (DR).
  -- It must be large enough to contain address, value, and control signals.
  -- For simplicity, we assume a 64-bit DR (to hold two 32-bit values or a complex
command)
  constant JTAG_DR_WIDTH : integer := 64;
  signal jtag_data_register_in  : std_logic_vector(JTAG_DR_WIDTH-1 downto 0);
  signal jtag_data_register_out : std_logic_vector(JTAG_DR_WIDTH-1 downto 0);
  -- Intermediate signals for debug_module operations
  signal s_wr_value  : std_logic_vector(31 downto 0);
  signal s_rd_addr   : std_logic_vector(JTAG_REG_ADDR_WIDTH-1 downto 0);
  signal s_wr_ena    : std_logic;
  signal s_wr_addr   : std_logic_vector(JTAG_REG_ADDR_WIDTH-1 downto 0);
begin
  -- Connect the TDO output of the JTAG
  -- The least significant bit of the DR is shifted out first
  jtag_tdo <= jtag_data_register_out(0);
  -- Process to manage the JTAG Data Register (DR)
  -- This is the conceptual equivalent of how data is exchanged via JTAG
  process (jtag_tck, jtag_test_logic_reset)
  begin
```

```
    if jtag_test_logic_reset = '1' then
      jtag_data_register_in  <= (others => '0');
      jtag_data_register_out <= (others => '0');
    elsif rising_edge(jtag_tck) then
      if jtag_shift_dr = '1' then
        -- Shift incoming data (tdi) into the DR
        jtag_data_register_in <= jtag_tdi & jtag_data_register_in(JTAG_DR_WIDTH-1
downto 1);
        -- Output data is handled outside this process
      elsif jtag_capture_dr = '1' then
        -- Load the data to be read (debug_rd_value) into the DR to be shifted out
        jtag_data_register_out <= debug_rd_value & std_logic_vector(to_unsigned(0,
JTAG_DR_WIDTH - debug_rd_value'length)); -- Load rd_value into the first 32 bits, rest to
0
      elsif jtag_update_dr = '1' then -- In Update-DR state
        -- When exiting Shift-DR and entering Update-DR, the shifted-in value
        -- in the DR is stable and can be used for operations
        jtag_data_register_out <= jtag_data_register_in;
      end if;
    end if;
  end process;
  -- Logic to decode commands from the JTAG Data Register and control the
debug_module
  process(PCLK, PRSTn)
  begin
    if PRSTn = '0' then
      s_wr_value  <= (others => '0');
      s_rd_addr   <= (others => '0');
      s_wr_ena    <= '0';
```

```
        s_wr_addr  <= (others => '0');
        debug_wr_value <= (others => '0');
        debug_rd_addr  <= (others => '0');
        debug_wr_ena   <= '0';
        debug_wr_addr  <= (others => '0');
      elsif rising_edge(PCLK) then
        -- Default to zero to avoid continuous operations
        debug_wr_ena <= '0';
        if jtag_update_dr = '1' then
          -- Decode data shifted into jtag_data_register_out
          s_wr_ena   <= jtag_data_register_out(0); -- Write control bit
          s_wr_addr  <= jtag_data_register_out(5 downto 1);
          s_wr_value <= jtag_data_register_out(37 downto 6);
          s_rd_addr  <= jtag_data_register_out(42 downto 38);
        end if;
        -- Load values into the outputs of the debug_adapter
        debug_wr_value <= s_wr_value;
        debug_rd_addr  <= s_rd_addr;
        debug_wr_ena   <= s_wr_ena; -- This should be a pulse, not a constant level
        debug_wr_addr  <= s_wr_addr;
        -- Reset or deactivate write enable after one cycle, or based on a flag
        -- to avoid continuous writes.
      end if;
    end process;
end architecture behavioral;
```

12.4 Altera virtual JTAG

Only this file was taken from the online repository [8].

12.5 Additional Technical Documentation from Droxygen

The complete code documentation, generated with Doxygen, is provided in the digital attachments (html/ folder).