École Polytechnique Fédérale de Lausanne

Real-time debugging of a soft-core OpenRISC CPU

by Jules Peyrat

Semester Project Report (v2)

Dr. Paolo Ienne
Project Advisor

Dr. Theo Ties Jan Kluter
Project Supervisor

IC LAP
INF 136 (Bâtiment INF)
CH-1015 Lausanne

January 5, 2026

Acknowledgements

I sincerely thank my project supervisor, Dr. Theo Kluter, for allowing me to work on this semester project within the LAP laboratory. I found great interest in this project, and I believe it will greatly support my academic and professional development.

Lausanne, January 5, 2026 Jules Peyrat

Abstract

The CS-473 course at EPFL uses the Gecko5Education board featuring a soft-core OpenRISC CPU for teaching system-level programming. However, the existing debugging interface designed for the OR1200 processor is not fully compatible with the virtual prototype used in this course, preventing real-time debugging and step-by-step execution.

This project aims at adapting the Advanced Debug Interface (ADI) to enable GDB-based debugging for the CS-473 CPU. On the software side, we implement a new OpenOCD driver to communicate with the Lattice ECP5 FPGA's native JTAGG hardware primitive, which differs from the soft-core TAP controllers typically used with the ADI. On the hardware side, we refactor the Bus Interface Unit (BIU) of the Wishbone module to interface with the virtual prototype's system bus, enabling memory read and write operations. Adapting the CPU module's BIU and connecting the CPU stall control signals is also required to access the register file, and perform step-by-step execution and breakpoint management.

Contents

Chapter 1

Introduction

The CS473 course at EPFL relies on a soft-core CPU virtual prototype of the OpenRISC 1000 CPU specifications, synthesized and executed on a Lattice ECP5 FPGA, enabling students to experiment with low-level architectural concepts in a realistic environment.

The first, public, Verilog implementation of the OpenRISC 1000 architecture—or *or1k*— is referred to as the *OR1200* CPU [7], for which debugging interfaces are already available [11] [3]. These interfaces support integration with the GNU Debugger (GDB) [2], allowing real-time debugging features such as CPU state inspection, step-by-step execution, memory reads and writes, and breakpoint management. Such capabilities are essential for efficient development, testing, and educational use.

However, the CPU used in CS-473 has several architectural differences from the OR1200 core. As a result, the existing OpenRISC debugging infrastructure cannot be used without modification.

1.1 Project Overview

The objective of this project is to enable real-time debugging capabilities for the soft-core CPU used in the CS-473 course by adapting an existing OR1200-compatible debug interface [11], which will be referred to as the *Advanced Debug Interface*, or ADI.

The core of this work consists of refactoring the ADI originally developed for the OR1200 processor. In order to allow for memory access through GDB, the memory submodule of the ADI must be adapted to the bus architecture employed on the virtual prototype (VP). In order to allow for register access, program breakpoints, and step-by-step execution, the CPU submodule of the ADI must be refactored.

On the hardware side, this project leverages the native JTAG primitive provided by the Lattice

ECP5 FPGA. Notably, this is the JTAG terminal used to reconfigure the FPGA gates when uploading a program onto the virtual prototype. The debug interface used in this project is designed to work with soft-core JTAG TAP controllers.

On the software side, OpenOCD only supports a subset of JTAG TAP controllers, while the native JTAG primitive of our Lattice FPGA is not one of them. Consequently, OpenOCD needs a new driver to communicate with the JTAG terminal of our Lattice FPGA.

The full code of the virtual prototype, testbenches and additional documentation can be found in the following **GitHub repository**. The code of the OpenOCD fork used in this project can be found in **this repository**.

1.2 Report Overview

The first chapter introduces important background on software used in this project, such as OpenOCD, which will be used as a hardware-software bridge to communicate with the ADI. It also presents the FPGA used in this project, as well as the JTAG standard, which is used to communicate with the board.

The second chapter provides key technical implementation details about the OpenOCD driver used to communicate with the debug controller, as well as the Verilog implementation of the debug controller itself.

The third chapter presents a few testbenches to validate several parts of the debug controller, and results of real-time tests executed on the synthesized design.

Chapter 2

Background

2.1 GDB

The *GNU Debugger* (GDB) [2] is a powerful and flexible debugging tool widely used in embedded systems development. In real-time debugging scenarios, GDB provides developers with the ability to inspect memory, analyze register states, and control program execution. When connected through a suitable debug interface—such as a JTAG-based bridge and a custom on-chip debug module—GDB enables non-intrusive, step-by-step investigation of running code while minimizing disruption to program execution.

Beyond real-time debugging, GDB is also a convenient tool for uploading programs directly to the FPGA hosting the soft-core processor. When paired with an appropriate debug stub or on-chip bootloader, GDB is able to load compiled binaries into the processor's memory space through the same physical interface used for inspection and control.

2.2 OpenOCD

OpenOCD (Open On-Chip Debugger) [4] is an open-source software tool designed to provide debugging, programming, and boundary-scan functionalities for embedded systems. It acts as a bridge between software development tools running on a host computer and target hardware. OpenOCD supports a wide range of processors, architectures and hardware interfaces (including JTAG).

In the context of this project, OpenOCD is used as a backend debug server interfacing with the FPGA through a JTAG connection. On one hand, it provides a Telnet server that allows direct interaction with the JTAG interface by issuing low-level debug commands (*raw* bit-level DR/IR scans). This mode is particularly useful for testing FPGA debug logic. On the other hand,

Signal	Name	Description
TCK	Test Clock	TAP clock signal
TMS	Test Mode Select	Used to navigate through the TAP FSM
TDI	Test Data In	Data input bitline
TDO	Test Data Out	Data output bitline

Figure 2.1: JTAG Synchronous Signals

OpenOCD exposes a GDB Remote Serial Protocol (RSP) server, enabling a direct connection between GDB and the target CPU. Through this interface, GDB can communicate with both the processor core and the system bus, allowing real-time debugging features such as register inspection, memory access, execution control, and breakpoint handling.

2.3 OpenRISC

OpenRISC is an open-source project aimed at developing a free and fully documented RISC instruction set architecture (ISA) and corresponding processor cores. The first publicly available implementation of the OpenRISC 1000 architecture is known as the OR1200, and features a complete, 5-stage pipelined, RISC-based processor with a debugging interface.

In this project, we work with a CPU that is closely based on the OR1200 core but introduces several adaptations to fit the virtual prototype platform used in the CS-473 course. Notably, the system bus architecture differs from the original OpenRISC Wishbone bus.

Note that the GDB build used in this work is the custom build provided in the `or1k-elf` toolchain [8].

2.4 The JTAG Standard

JTAG (*Join Action Test Group*) is a hardware communication protocol mainly used for debugging system-on-chips in real-time. The JTAG bus is composed of four synchronous signals and an optional asynchronous active-low reset signal. The four synchronous signals are presented in figure 2.1.

TAP (*Test Access Port*) controllers are hardware components that software JTAG drivers/bridges communicate with. TAP controllers implement the JTAG TAP finite state machine presented in figure 2.2.

JTAG TAPs all have one internal register called the **instruction register**. Its size varies with the number of instructions the TAP controller supports. In order for the computer to send payload

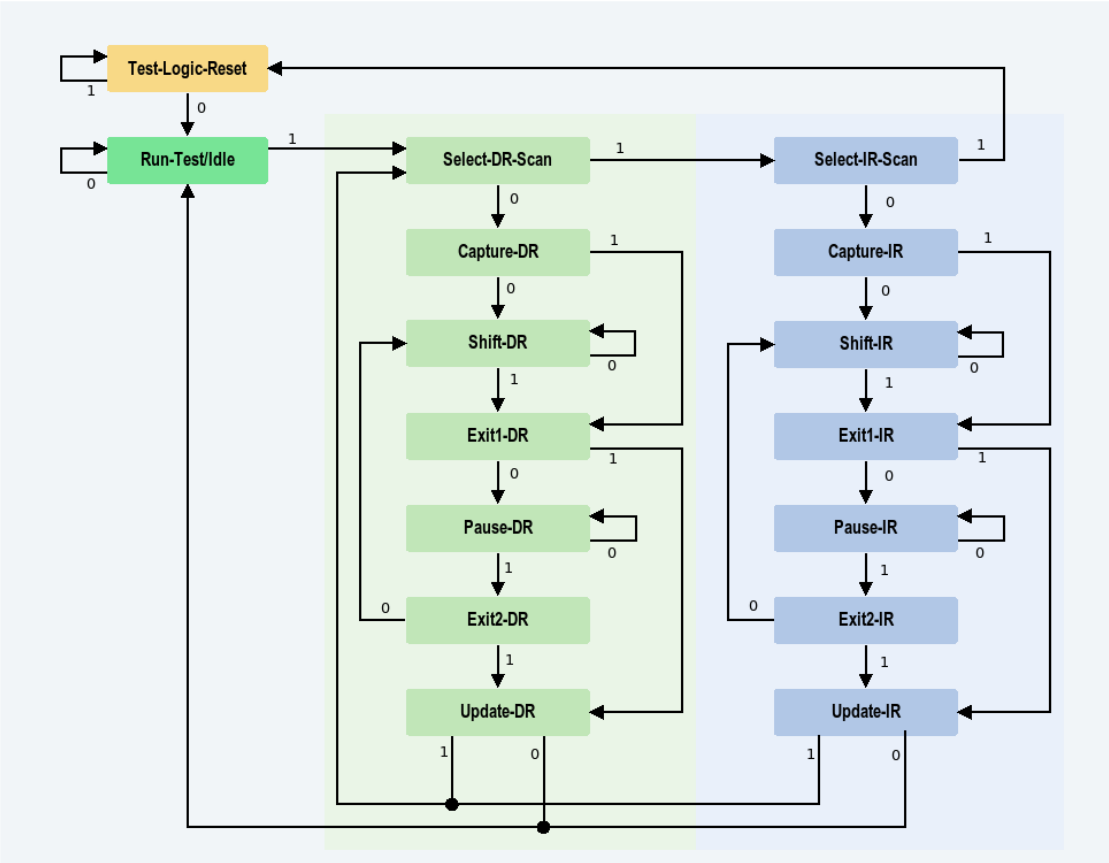


Figure 2.2: JTAG TAP Finite State Machine

```
-- Instruction register description
        attribute INSTRUCTION_LENGTH of LFE5U_85F_XXBG756 : entity is 8;
        attribute INSTRUCTION_OPCODE of LFE5U_85F_XXBG756 : entity is
                "              IDCODE                (11100000)," &
                "          ISC_ENABLE                (11000110)," &
                "    ISC_PROGRAM_DONE                (01011110)," &
[...]
                "           ISC_ERASE                (00001110)," &
                "            ISC_NOOP                (00110000)," &
                "              SAMPLE                (00011100)," &
                "               HIGHZ                (00011000)," &
                "              EXTEST                (00010101)," &
```

Figure 2.3: Lattice ECP5 BSD File Snippet

data (test vector, command...) to the TAP controller, it first needs to load the correct instruction operation code inside the instruction register through an **IR-scan**. The *opcode* stored inside the IR determines the data register exposed to the computer during subsequent **DR-scans**.

TAPs all share at least two instructions: **IDCODE** and **BYPASS**. When **BYPASS** is loaded inside the IR, the TDI and TDO lines are connected. This is useful for selectively disabling some TAPs when multiple TAPs are connected in series on the same JTAG chain: this way, the JTAG software bridge can communicate with only a subset of the in-series TAPs. The **IDCODE** instruction allows returning the 32-bit identification code of the debugged SoC.

While **IDCODE** and **BYPASS** are generic instructions, TAPs all have instructions that are specific to the FPGA/target architecture. Such instruction codes can be found in the BSD (*Boundary Scan Description*) files of the target architecture. Depending on the number of instructions available on the TAP, the size of the IR may vary. For instance, the Lattice ECP5 FPGA has an *irlen* of 8, and instructions **ISC_PROGRAM_DONE** (`0b01011110`) and **ISC_ERASE** (`0b00001110`) are used to reprogram the FPGA gates.

2.5 Lattice ECP5 FPGA

2.5.1 Gecko5Education

The *Gecko5Education* board is an FPGA development platform primarily used for teaching digital systems and computer architecture at EPFL. It is designed to give students hands-on experience and features a **Lattice ECP5 FPGA** at its core. The board also includes various peripherals suitable for system-on-chip experimentation, such as DRAM memory blocks, General-Purpose I/Os (GPIOs) like push buttons and switches, an RGB LED matrix, and a UART interface for serial

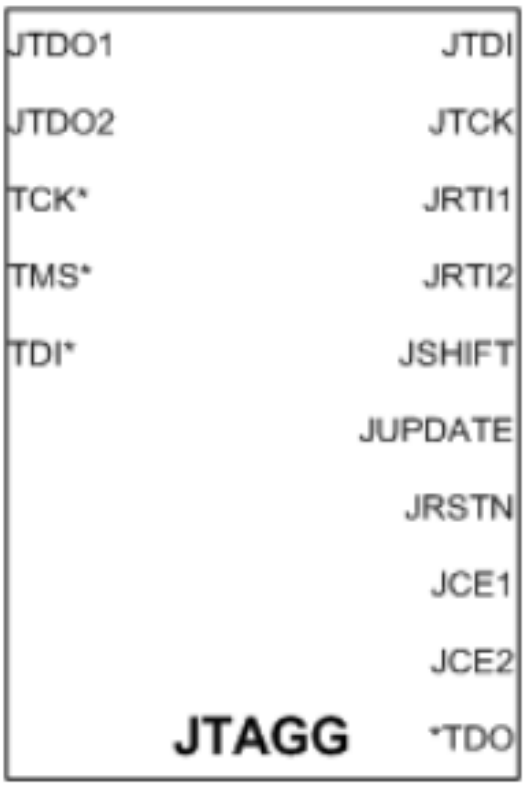


Figure 2.4: JTAGG Primitive

communication.

Although the board provides a JTAG port through a USB-C connector, it was previously unused due to its minimal implementation and both the availability and ease-of-use of UART for communication. Unlike CPUs, which execute software instructions sequentially, FPGAs are reconfigurable hardware platforms used to implement custom digital logic circuits. They are programmed using hardware description languages (HDLs); in this project, Verilog is used to describe the hardware's structure and behavior. The HDL code is then synthesized, placed, and routed to produce a bitstream, which is loaded into the FPGA to configure its logic.

2.5.2 The JTAGG Primitive

The **JTAGG** [9] component is the hardware implementation of the JTAG TAP Controller available on Lattice ECP5 FPGAs. When working on the Gecko5Education, the JTAGG TAP is used to reconfigure the FPGA logic using tools like *openFPGALoader*.

For security, the JTAGG primitive does not provide direct access to the four main raw JTAG

Signal	Description
JTDI	Latched (one clock cycle late) TDI signal
JTCK	Same as TCK
JRTIx	Active high when TAP in Run-Test/Idle state
JSHIFT	Active high when TAP in Shift-DR
JUPDATE	Active high when TAP in Update-DR
JRSTN	Active low, async reset line
JCEx	Active high when TAP in Shift-DR or Capture-DR
JTDOx	User instruction output (TDO)

Figure 2.5: JTAGG Synchronous Signals

signals (*TCK*, *TMS*, *TDI* and *TDO*) and only provides access to signals presented in figure 2.5. The JTAGG primitive provides two slots for user-defined instructions with respective opcodes `0x32` and `0x38`. In this project, we use opcode `0x32` to interface with our internal debug controller component. Note that *JRTI*`x` and *JCE*`x` are only high when the corresponding user instruction is loaded into the IR. `x` can be either 1 or 2.

2.6 Advanced Debug System

The *Advanced Debug System* [11] project is an open-source soft-core available on OpenCores. It provides several components, including an **Advanced Debug Interface** (or "ADI"), which is the component between the JTAG controller and the soft-core SoC, as well as several implementations of **soft-core JTAG TAP controllers** such as `xilinx`, `mohor`, `actel` or `altera`. We discuss the structure of the ADI in the following chapter.

According to its documentation, the usual approach is to synthesize a soft-core TAP controller alongside the ADI and to use four FPGA physical pins to emulate a standard JTAG connection. The documentation also mentions the possibility of directly using the hardware JTAG controller already present on the FPGA—typically the one used for gate array reconfiguration—but this option is limited to devices that implement the UJTAG interface, such as those in the ProASIC family.

In this project, we take a slightly different approach: we directly leverage the native JTAG hardware controller available on our Lattice FPGA. As a result, the provided soft-core TAP implementations are used only as reference designs to guide the integration of the Advanced Debug Interface, rather than being instantiated in our final system. Furthermore, we make use of the reference soft-core implementation of the JTAGG primitive provided by Tom Verbeure [10] to help with our ADI refactor.

While using a soft-core TAP controller has the benefit of exposing all raw JTAG signals, using the hardware JTAG primitive of our FPGA along with the existing USB terminal avoids having

to set up an additional USB terminal from scratch. In the following chapter, we discuss the modifications required to the ADI to adapt to this non-native TAP controller.

Chapter 3

Implementation

3.1 OpenOCD

3.1.1 OpenOCD Driver

In order to communicate with our debug controller, OpenOCD needs to know:

- the DR register sizes (or *DR length*) and command formats,
- the IR register size (or *IR length*),
- the IR opcodes that correspond to our debug controller.

Since our processor is very similar to the OR1200 processor, in order to avoid having to implement both a software driver and a hardware debug controller entirely from scratch, we use one of the OR1200-ready debug interfaces supported by OpenOCD [3] [11], namely the ADI (*Advanced Debug Interface*). Consequently, the DR register sizes and command formats remain exactly the same, as they are embedded in the semantics of our ADI driver. The OpenOCD source file of our ADI driver is `or1k_du_adv.c`, and no modification is required.

Since we are using the JTAGG hardware primitive of our Lattice FPGA, we have no choice but to use the JTAGG IR length, which is 8. This number is provided to OpenOCD inside the OpenOCD server config file through the `-irlen` parameter (see figure 3.1).

The IR opcodes depend on the TAP controller that we are communicating with. The ADI actually provides three different Verilog implementations of popular TAP controllers, including *Mohor*, *Xilinx* and *Intel*. The user is free to choose which TAP controller to implement alongside the ADI. All of these TAP controllers are supported by the OpenOCD *or1k* driver, and it is the

responsibility of the user to specify in the OpenOCD server configuration file which TAP controller is being used by the ADI through the `tap_select` command (see figure 3.1). OpenOCD has three different JTAG TAP drivers, one for each of these controllers, all of them being mutually incompatible with one another.

However, none of these drivers are compatible with the Lattice JTAGG primitive. Consequently, we need to write a new OpenOCD driver that will support it. Fortunately, the amount of changes required relative to the existing *Mohor/OpenCores* TAP driver is very limited. We start with the *Mohor/OpenCores* TAP driver (whose source file is `or1k_tap_mohor.c`) and perform the following changes:

- increase its IR length from `4` to `8`,
- change the IR opcode of the debug controller from `0x8` to `0x32` to match the first JTAGG user instruction code.

We compile OpenOCD along with this new driver of ID `gecko`, stored in the `or1k_tap_gecko.c` source file. This driver can be activated using the command `tap_select gecko` inside an OpenOCD server configuration file (see figure 3.1).

3.1.2 Additional Notes

Note that the native ADI driver has a small bug when a Wishbone bus error happens while performing a bus operation. Normally, if the driver observes a Wishbone bus error during its transaction, it should retry up to 3 times before giving up. The bug causes OpenOCD to loop and retry the memory operation forever (the user then has to manually kill the process). This bug is not very convenient when working on the BIU, as any mistake made with system bus protocol would often result in a bus error, causing OpenOCD to freeze. This bug has been fixed in the provided OpenOCD source.

OpenOCD also has a bug with legacy/hi-speed support, more information about this in section 3.3.3.

3.1.3 Server Config File

In the OpenOCD server config presented in figure 3.1, we provide the IDCODE and IR length of the Lattice FPGA TAP controller. Then, we associate this TAP controller to a big-endian *or1k* target CPU. We select our TAP driver and debug unit interface. Recall that OpenOCD does support two debug interfaces [3] [11], one being more recent and performant than its ancestor.

```
set _CPUTAPID  0x41113043
set _ENDIAN big
set _TARGETNAME $_CHIPNAME.cpu
set _CHIPNAME or1k

jtag newtap $_CHIPNAME cpu \
    -irlen 8 -irmask 0x83 -ircapture 0x1 \
    -expected-id 0x41113043
target create $_TARGETNAME or1k \
    -endian $_ENDIAN \
    -chain-position $_TARGETNAME
tap_select gecko
du_select adv [expr {0}]
```

Figure 3.1: OpenOCD Server Config

If we were to only register a "raw" TAP controller in OpenOCD, we could connect to the OpenOCD server and send raw JTAG commands through a Telnet client. By associating this TAP controller to an *or1k* CPU, we tell OpenOCD to interpret this JTAG terminal as a debuggable CPU. As a result, it opens a GDB RSP server instead, that we can connect to using GDB.

3.1.4 Connecting using Telnet or GDB

The following command can be used to open a Telnet connection to the OpenOCD server (running on port 4444) to send raw JTAG commands to the ADI:

```
telnet localhost 4444
```

The following command can be used to open a connection to the GDB RSP server (running on port 3333) to debug the CPU using GDB.

```
or1k-elf-gdb hello.elf --eval-command='target remote localhost:3333'
```

3.2 Top-Level Debug Controller

The `adv_debug_if` debug controller is composed of a top-level debug controller with four sub-modules (see figure 3.2): two sub-modules for the CPU cores, one sub-module for accessing the Wishbone system bus, and one sub-module for the JSP server. *The JSP server sub-module is out of the scope of this project—it allows emulating a serial connection through our JTAG connection.*

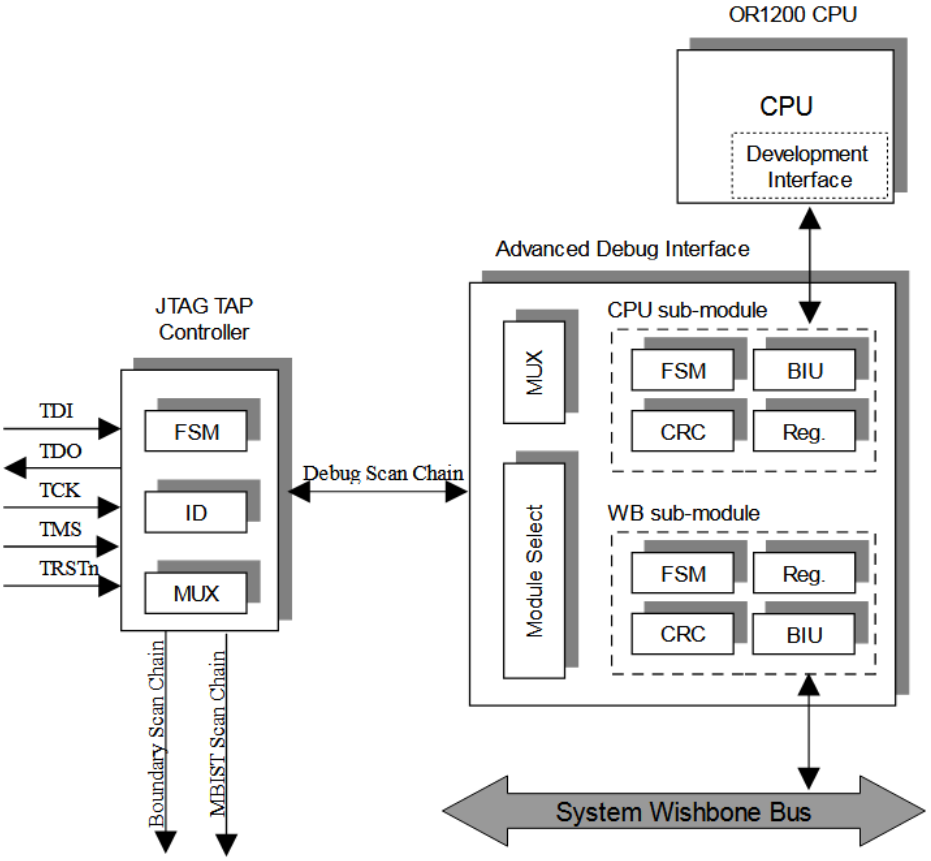


Figure 3.2: ADI Global Structure (Source: [12])

The top-level debug controller receives 53-bit commands from the JTAG TAP. The MSB indicates if the JTAG command is directed to the top-level debug controller. There is only one type of command for the top level, namely sub-module selection. Since there are four sub-modules, we need two bits to uniquely identify all sub-modules. The **Wishbone** sub-module has ID `0b00` while **CPU0** and **CPU1** sub-modules have IDs, respectively, `0b01` and `0b10`. Therefore, the sub-module selection command is only three bits long.

Even though the DR input register of the top-level ADI is 53 bits long, OpenOCD actually only needs to shift in 3 bits. When communicating with the top-level, the MSB is always set to 1, indicating a module selection command. This bit is followed by the 2 bits identifying the module to be selected. If the MSB is set to 0, then the command has to be 53 bits long exactly, and it will be dispatched to the currently selected submodule.

3.2.1 TDI 1-Cycle Delay Problem

In order to avoid long critical paths, the JTAGG primitive latches the input signal TDI, making the JTDI signal one cycle late. If no modification is done to the current implementation of the debug controller input shift register logic, the LSB of the JTAG command will be garbage, and the MSB will be lost.

In the previous semester project [1] focusing on JTAG support for Gecko5Education, the solution proposed was to virtually increase by 1 bit all DR-scans sent to the JTAGG TAP controller. This has the effect of erasing the garbage LSB and loading in the MSB at the last moment in the input shift register. While this solution works, it is not applicable to our situation without major OpenOCD refactors. A hardware-side patch would be much more convenient as it would allow our debug controller to be compatible with the existing *or1k* OpenOCD driver.

To patch this, we delay the internal `shift_dr_i` signal by one cycle. Recall that this signal indicates if the TAP is currently in state *Shift-DR*, in which case the debug controller needs to shift its internal input shift register by one bit, while loading one bit from the TDI line and exposing the outgoing LSB to the TDO line. This patch has the effect of "waiting" for the relevant signal to be ready on the TDI line.

Fortunately, **there is exactly a 1-cycle time frame** between the last bit of payload sent through TDI and the time by which the TAP FSM enters the *Update-DR* state. This is due to the fact that the TAP FSM needs two cycles with **TMS** set to **1** to exit the *Shift-DR* state and enter the *Update-DR* state. Since all internal register updates of the debug sub-modules happen on this last state, this leaves us exactly the amount of time needed to fix the content of the input shift register.

3.3 System Bus Module

The system bus sub-module, or *Wishbone* module, with module ID `2'b0`, is responsible for interfacing between the ADI and the system bus. Since the SDRAM controller is connected to the system bus, it is the module GDB uses to perform memory reads and writes.

When OpenOCD receives a memory read (or write) requested by the user through GDB, this memory read (or write) is interpreted as two distinct DR-scans. The first DR-scan is always a *Burst Setup* command, and the second scan is always a *Burst* command.

Intuitively, the first DR-scan provides the address to read from (or write to), the number of bytes to read (or write), and the word size. It is always of fixed size (53 bits), and its structure is presented in figure 3.3a. The *Burst Setup* format is the same for reads and writes. The second DR-scan executes the memory read (or write) by either reading words from the Bus Interface Unit (BIU) and transferring them through the JTAG TDO line, or by shifting in bits from the TDI line and instructing the BIU to write full 32-bit words.

The BIU is the subcomponent of the Wishbone module responsible for interfacing with the SoC system bus. Since our SoC does not use the native ADI bus architecture (Wishbone), most of our work consists of refactoring the state machine of the BIU to match our SoC system bus architecture.

3.3.1 Memory Read Example

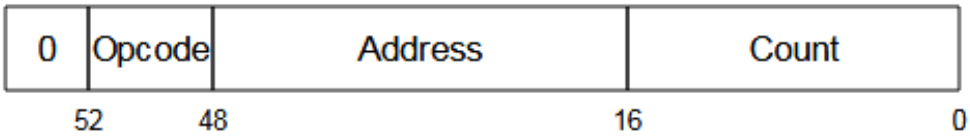
The corresponding *Burst Setup* command for reading 4 bytes of memory from address `0xABC` is the following. Note that according to the ADI documentation [12], opcode `0x7 = 0b111` corresponds to a 32-bit read.

```
0b0_0111_00000000000000000000101010111100_0000000000000100
```

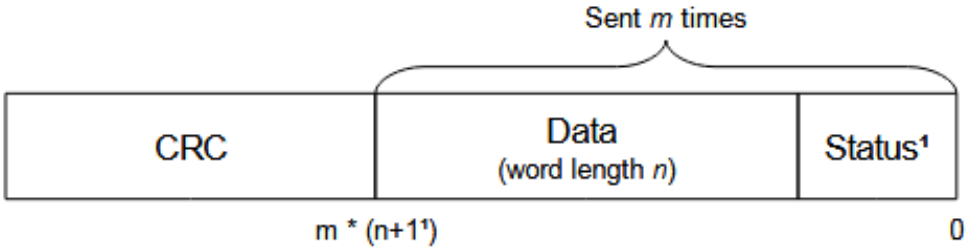
After sending the *Burst Setup* command, OpenOCD will perform a *Burst Read* command by shifting in only 0's. The ADI does not care about the bits shifted in, as the only important part is the bits of payload shifted out through TDO. The format of *Burst Read* commands is presented in figure 3.3b. One status bit (see section 3.3.3) indicates if the memory operation was performed successfully. For instance, it is unasserted if a bus error happened during the transaction, instructing OpenOCD to try the same transaction again. This status bit is followed by the data payload (32-bit words), followed by a 32-bit CRC code for error detection.

In case of a write, the *Burst Setup* command is very similar. The opcode is now `0x3` for 32-bit writes.

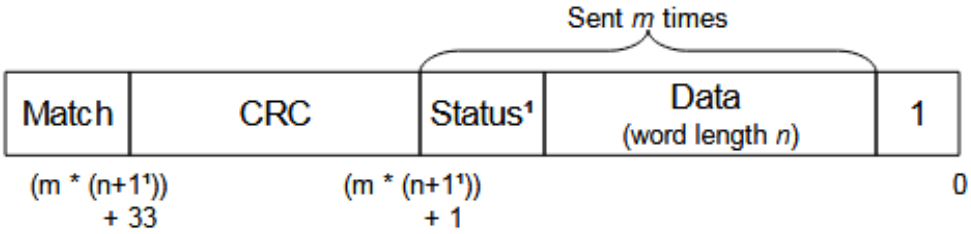
```
0b0_0011_00000000000000000000101010111100_0000000000000100
```



(a) Burst Setup



(b) Burst Read



(c) Burst Write

Figure 3.3: OpenOCD JTAG Command Formats (Source: [12])

The *Burst Write* command format is presented in figure 3.3c. This time, OpenOCD shifts in the 32-bit words that must be written to memory, potentially interleaved with status bits (see section 3.3.3), followed by a 32-bit CRC. The last bit transferred is a *match* bit, which the ADI uses to indicate to OpenOCD if the CRC transferred by OpenOCD matches the one computed in hardware by the ADI. In case of a CRC error, the transaction is aborted. OpenOCD listens to this *match* bit as well as *status* bits to make sure no bus error occurred during the transfer.

3.3.2 FSM Refactoring

The FSM of the initial Wishbone BIU only has two states, `IDLE` and `TRANSFER`. In the `IDLE` state, the BIU has either 1) never received any instruction from the Wishbone submodule, or 2) has successfully performed the required memory operation, and in case of a read, the read payload is available in internal register `data_out_reg`, or 3) has performed the memory operation which resulted in an error, in which case the `err_reg` internal register is asserted.

The Wishbone standard specifies two important signals that allow the BIU FSM to switch

between states. wb_strobe_i is asserted when a transaction starts: the FSM transitions from IDLE to TRANSFER. When wb_ack_i is asserted, it indicates that the memory operation was successfully conducted, and the FSM can successfully transition back to IDLE. In case of a Wishbone bus error, wb_err_i is asserted.

The major difference in our new FSM is that the BIU now needs to wait for a grant from the bus arbiter. The system bus is used by multiple peripherals simultaneously, and access to the system bus is granted by a central bus arbiter. The major FSM change is the addition of a WAIT state. The BIU transitions to this state when the Wishbone submodule instructs a new memory operation. It asserts a request signal, and waits for a grant from the arbiter. When obtaining this grant, it transitions to the TRANSFER state. In case of a read, the BIU provides the memory address being read from and waits for the sb_data_valid_i signal, which is the equivalent of the wb_ack_i signal.

In case of a write, our system bus behaves slightly differently. The VP system bus has a sb_busy signal that indicates that the bus slave has not been able to execute the memory write provided on the last clock cycle. Consequently, the FSM needs to stay in the TRANSFER state until the first cycle where the busy bit is zero. On the first cycle where the busy bit is unasserted, our BIU transitions back to the IDLE state. By design, the BIU is never busy, so the read operation is marked successful as soon as the data_valid signal is asserted.

3.3.3 Hi-Speed Mode

This section discusses an issue encountered while refactoring the original ADI. The OR1200 ADI has two modes for memory burst commands: *Legacy* and *Hi-Speed* mode. In legacy mode, memory read/write bursts transfer an additional status bit for every word transferred over JTAG. In hi-speed mode, memory bursts only provide exactly one status bit at the beginning of the burst DR-scan.

The user can choose which mode to use by flipping a constant in the RTL specification of the ADI before synthesis. In any case, the option specified in the ADI RTL must match the option provided in the OpenOCD server configuration file, as it alters the JTAG command formats used.

The reason for this status bit is the following. Suppose we are performing a memory read from the system bus. Recall that the Wishbone BIU implements a basic "best-effort optimistic" protocol for reading from and writing to the system bus. For memory reads, for instance, while the bits of the previous 32-bit words are transferred through the JTAG line, the BIU requests access to the system bus, waits for a grant from the arbiter, and performs the next single 32-bit word read as soon as bus grant is provided.

In other words, if the system bus is heavily congested, the BIU may not have completed the next 32-bit read by the time the Wishbone top-level module expects to shift out valid data. This is precisely why status bits are included in the returned data. Each status bit is set to 1 if the BIU

was ready—that is, if the corresponding single-word read on the system bus actually took place. Conversely, if the status bit is 0, it indicates that the BIU had not yet completed the read, and therefore the bits shifted out during that cycle must be considered garbage.

The *or1k* OpenOCD driver is aware of these status bits, and will retry the full memory read until all status bits are set to 1, up to a user-defined threshold.

Most of the time, the SoC bus clock (in our case, synchronous) is much faster than the JTAG clock, which makes the JTAG TDI/TDO line the bottleneck of our JTAG data transmission. In our case, the JTAG controller operates at 4 MHz while the SoC operates at almost 80 MHz. If we only perform 32-bit words, this leaves the BIU `32 * 80 / 4 = 640` bus cycles to perform the next word read. For this reason, Hi-speed mode just optimistically presumes that the BIU never fails to execute bus transactions in time.

That being said, while the OpenOCD documentation explicitly states the possibility of disabling Hi-speed mode through the use of a config file option, the source code of the ADI driver never modifies the number of bits sent through JTAG depending on the option chosen. **It seems like the OpenOCD ADI driver is hard-coded to Hi-speed mode.**

While I planned to use Legacy mode for debugging (primarily for making sure that the system bus was executing bus operations fast enough), I was eventually forced into switching to Hi-speed mode for prototyping as it seemed like Legacy mode was simply not supported at all.

3.4 CPU Module

The CPU submodule of the ADI is responsible for reading and writing to the CPU register file, step-by-step execution, and handling of program breakpoints.

3.4.1 Register Read and Writes

The CPU submodule is structurally very similar to the Wishbone submodule, as presented in figure 3.2. It contains FSM logic, internal registers (see next section), as well as a CRC and a BIU. The BIU is the same as the Wishbone BIU, the only difference is that the Wishbone submodule communicates with the system bus, while the CPU submodule BIU communicates with the register file.

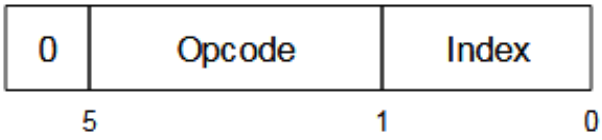


Figure 3.4: *Register Select* Command (opcode must be set to **0xD**)

3.4.2 Future Work

While step-by-step execution and program breakpoint management were in the scope of this project, I did not have time to start working on these real-time debugging features.

All ADI submodules (CPU, Wishbone, JSP) have internal module registers that can be read from or written to. The CPU submodule contains a single internal register, the *status register*, which is a 2-bit register. One of these bits indicates whether or not the CPU is currently stalled. Writing to this internal register stalls the CPU. Internal register reads and writes consist of two distinct DR scans. The first DR scan is a *Register Select* command as presented in figure 3.4. The second DR scan is similar to a burst read. OpenOCD needs to shift in at least as many bits as the size of the internal register. The bits shifted out by the ADI correspond to the current internal register content.

Chapter 4

Evaluation

4.1 Real-Time Tests

4.1.1 Sub-Module Selection

To test sub-module selection command, we display the ID of the selected module on the RGB LED array of the Lattice FPGA.

We use a Telnet connection to connect to OpenOCD, and send DR-scans of length 3 to select a module. The module ID is stored in an internal register called `module_id_reg`. The commands `irscan ecp5.tap 0x32` followed by `drscan ecp5.tap 3 0b110` select the **CPU1** submodule. The second leftmost LED lights up when `module_id_reg // 2 == 1`, the leftmost lights up when `module_id_reg % 2 == 0`.

This simple test allows us to properly test our interfacing with the JTAGG component and make sure that the 1-cycle-late-TDI problem is properly patched.

4.1.2 Register Reads

While register reads and writes were not implemented in this project, we can still validate the JTAG and ADI integration by trying to read "dummy" values from the register file. This can be done very easily by hard-coding the `rdy_o` signal to `high`, and the `data_out_reg` signal to a dummy value like `0xDEADBEEF`. This tricks the CPU submodule into thinking the BIU has successfully performed its last instructed memory read.

The GDB command `p/x <register>` can be used to display the content of a general-purpose register from the *or1k* register file. The output is the following, which confirms that 1) the top-

level debug controller successfully received and dispatched the read command to the CPU submodule and 2) the CPU submodule read data from its BIU.

```
(gdb) p/x $r2
$3 = 0xdeadbeef
```

Note that after performing at least one memory operation, OpenOCD cycles indefinitely through the CPU and Wishbone modules. I believe this is due to OpenOCD trying to stall the CPU for step-by-step execution, but since this has not been implemented yet, OpenOCD keeps trying indefinitely.

```
Debug: 409 11536 or1k_du_adv.c:263 adbg_select_module(): Select module: CPU0
Debug: 410 11537 or1k_du_adv.c:263 adbg_select_module(): Select module: WISHBONE
Debug: 411 11636 or1k_du_adv.c:263 adbg_select_module(): Select module: CPU0
Debug: 412 11637 or1k_du_adv.c:263 adbg_select_module(): Select module: WISHBONE
....
```

4.1.3 Memory Reads

The GDB command `x/wx <address>` can be used to read a 32-bit word at a given address in memory. The resulting OpenOCD debug logs are shown in figure 4.1 for 32-bit aligned and unaligned reads.

The top figure presents a 4-byte aligned 32-bit (full word) memory read. Since it is aligned, it is properly interpreted as a single burst read on the bus. On the bottom figure however, we can see than the OpenOCD *or1k* driver splits up an unaligned 32-bit read into three distinct reads: two 1-byte reads at `0x401` and `0x404` with `byte_enables` masks of `0b0100` and `0b1000` respectively, and one 2-byte read at `0x402` with mask `0b0011`.

This is done accordingly to the memory alignment specifications provided in the OR 1000 architecture manual [6]. Indeed, the 32-bit OR 1000 architecture does not support unaligned 4-byte (full word) reads.

4.1.4 Memory Writes

The GDB command `set {int}0x<address> = <value>` can be used to write a 32-bit word value to a specific address in memory. The output of such command is provided on figure 4.2.

```
Debug: 361 [or1k.cpu] {1} received packet: m400,4
Debug: 362 addr: 0x0000000000000400, len: 0x00000004
Debug: 363 reading buffer of 4 byte at 0x00000400
Debug: 364 Read memory at 0x00000400, size: 4, count: 0x00000001
Debug: 365 Reading WB32 at 0x00000400
Debug: 366 Select module: WISHBONE
Debug: 367 Doing burst read, word size 4, word count 1, start address 0x00000400
Debug: 370 CRC OK!
Debug: 372 [or1k.cpu] {1} sending packet: $a2aa2aae#ae
```

(a) Aligned 32-bit Memory Read

```
Debug: 1284 [or1k.cpu] {1} received packet: m401,4
Debug: 1285 addr: 0x0000000000000401, len: 0x00000004
Debug: 1286 reading buffer of 4 byte at 0x00000401
Debug: 1287 Read memory at 0x00000401, size: 1, count: 0x00000001
Debug: 1288 Reading WB8 at 0x00000401
Debug: 1289 Doing burst read, word size 1, word count 1, start address 0x00000401
Debug: 1292 CRC OK!
Debug: 1294 Read memory at 0x00000402, size: 2, count: 0x00000001
Debug: 1295 Reading WB16 at 0x00000402
Debug: 1296 Doing burst read, word size 2, word count 1, start address 0x00000402
Debug: 1299 CRC OK!
Debug: 1301 Read memory at 0x00000404, size: 1, count: 0x00000001
Debug: 1302 Reading WB8 at 0x00000404
Debug: 1303 Doing burst read, word size 1, word count 1, start address 0x00000404
Debug: 1306 CRC OK!
Debug: 1308 [or1k.cpu] {1} sending packet: $44004546#9b
```

(b) Unaligned 32-bit Memory Read

Figure 4.1: Memory Reads

```
Debug: 356 addr: 0x400, len: 0x00000004
Debug: 357 writing buffer of 4 byte at 0x00000400
Debug: 358 Write memory at 0x00000400, size: 4, count: 0x00000001
Debug: 359 Writing WB32 at 0x00000400
Debug: 360 Select module: WISHBONE
Debug: 361 Doing burst write, word size 4, word count 1,start address 0x00000400
Warn : 362 CRC ERROR! match bit after write is 0 (computed CRC 0xe5a59fe0)
Warn : 363 CRC ERROR! match bit after write is 0 (computed CRC 0xe5a59fe0)
Warn : 364 CRC ERROR! match bit after write is 0 (computed CRC 0xe5a59fe0)
Debug: 365 Reporting -4 to GDB as generic error
Debug: 366 [or1k.cpu] {1} sending packet: $E0E#ba
```

Figure 4.2: Writing a full word to memory

At the moment, memory writes still suffer from CRC errors. A plausible cause is the reoccurrence of the issue described in 3.2.1. While this issue was fixed for the top-level input register, the Wishbone submodule and its own CRC submodule still utilize this input line.

Testbenches do reproduce this issue, as the data stored in the `data_in_reg` (internal register that stores data to be written to the system bus) does not match the data written by OpenOCD. Then, since the CRC module inputs utilize the TDI input line, these need to be refactored too. The testbench provided does not actually check the CRC or data provided back to OpenOCD through the TDO line.

Burst Setups function correctly because they are first stored in the top-level input shift register (where TDI timing is corrected), then forwarded to the corresponding submodule. *Burst Reads* function properly as they do not utilize the TDI input line, unlike *Burst Writes* which use the TDI line to read data payload and compute CRC.

4.2 Testbenches

Testbenches are provided to test both top-level module selection and memory operations. These testbenches are stored in the folder `src/vp/modules/jtag/test`. The following testbenches are provided:

- `./tb_select_module.sh` to test top-level submodule selection
- `./tb_mem_read.sh` to test Wishbone read/writes.

```
(gdb) load
Loading section .vectors, size 0x184 lma 0x0
Loading section .text, size 0x200c lma 0x184
Loading section .rodata, size 0x16c lma 0x2190
Loading section .eh_frame, size 0x54 lma 0x22fc
Start address 0x00000100, load size 9040
Transfer rate: 420 KB/sec, 2260 bytes/write.
```

Figure 4.3: Uploading a program through GDB (4MHz JTAG)

4.3 Benchmarks

4.3.1 Upload Speed

One key feature of GDB coupled with a debug interface is the ability to upload programs to the target platform. Currently, in order to upload a program to the virtual prototype, we need to use a Serial graphical interface such as CuteCom to write the program directly into RAM over a serial UART connection. As Serial UART connections are known to be particularly slow, we would like to use JTAG instead to benefit from higher upload speeds. The `load` GDB command can be used to upload the provided program to the target RAM. The program to be uploaded is the one provided in the GDB run command (see section 3.1.4).

Figure 4.3 shows the output of a `load` command with CRC checks commented out in OpenOCD. Due to a bug in the current implementation of the debug interface, writes do not work yet (and yield CRC errors). However, it is safe to assume that disabling CRC checks has very limited influence on software overhead (and therefore benchmark results). Program upload speed depends on the JTAG clock, ADI mode (hi-speed or legacy), bus availability, and OpenOCD software overhead.

Using GDB and the debug interface for uploading programs, with a JTAG clock of 4MHz, we achieve **a program upload speed of 420KB/s**. We can compare this value to our Serial connection with baud rate 115200, which translates to at most 15KB/s with no overhead. The same "Hello World" program takes < 0.1s to upload using GDB and $\approx 1.8s$ to upload using CuteCom.

Note that it is not particularly useful to further crank up the JTAG clock speed, as upload latencies are already very limited, and because our BIU optimistic design assumes that our system bus is always available on time. Increasing the JTAG clock could break this assumption and lead to OpenOCD having to try memory writes again.

Chapter 5

Future Work

While the current implementation provides a foundation for real-time debugging of our *or1k* CPU, several aspects remain to be addressed to achieve full feature completeness.

Regarding memory accesses, the Bus Interface Unit (BIU) was refactored to ensure compatibility with the system bus of the virtual prototype. The lack of bus errors and matching CRC codes for reads indicate that the top-level ADI and JTAG were successfully interfaced. At the moment, writes still produce CRC errors due to a mishandling of the TDI input line inside the Wishbone module. Future work should focus on fixing the Wishbone module and validate program uploading.

Second, GPR access through the debug interface is not yet fully supported. The BIU within the CPU submodule must still be adapted to connect to the register file. The register file is currently implemented as a simple dual-port memory, with both ports already allocated to the CPU decode stage. One possible solution would be to duplicate the register file in order to provide additional read ports dedicated to debugging purposes.

Finally, advanced debugging features such as single-step execution and breakpoint management have not yet been implemented. These mechanisms require tight integration with the CPU control logic and pipeline stages, and represent a natural continuation of this work.

Chapter 6

Conclusion

Given its similarity to the OR1200 CPU, it is possible to implement real-time debugging for our soft-core CPU with limited modifications to our CPU, the *Advanced Debug Interface*, and OpenOCD drivers. While this work focuses on refactoring the BIUs of the CPU and Wishbone modules to adapt them to the virtual prototype SDRAM and register file, some refactoring work is still needed inside the ADI source code to fully support step-by-step execution, register content access, and program breakpoint support.

Bibliography

[1] Antoine Colson. *JTAG Support for Gecko5Education*. URL: https://github.com/nosloc/JTAG_support_for_Gecko5Education.

[2] GNU. *GDB: The GNU Project Debugger*. URL: https://sourceware.org/gdb/.

[3] Igor Mohor. *SoC Debug System*. URL: https://opencores.org/projects/dbg_interface.

[4] OpenOCD. *Open On-Chip-Debugger*. URL: https://openocd.org/.

[5] OpenRISC. *Debugging OpenRISC on FPGAs using GDB*. URL: https://openrisc.io/tutorials/docs/Debugging.html.

[6] OpenRISC. *OpenRISC 1000 - Architecture Manual*. URL: https://raw.githubusercontent.com/openrisc/doc/master/openrisc-arch-1.0-rev0.pdf.

[7] OpenRISC. *OR1200 Source Code*. URL: https://github.com/openrisc/or1200.

[8] OpenRISC. *or1k-elf Toolchain Releases*. URL: https://github.com/stffrdhrn/or1k-toolchain-build/releases.

[9] Lattice Semiconductors. *Lattice FPGA Reference Guide*. URL: http://www.latticesemi.com/view_document?document_id=52656.

[10] Tom Verbeure. *Lattice ECP5 User JTAG*. URL: https://github.com/tomverbeure/ecp5_jtag.

[11] Nathan Yawn. *Advanced Debug Interface*. URL: https://opencores.org/projects/adv_debug_sys.

[12] Nathan Yawn. *Advanced Debug Interface Documentation*. URL: https://github.com/freecores/adv_debug_sys/blob/master/Hardware/adv_dbg_if/doc/AdvancedDebugInterface.pdf.