Faculty of Engineering Sciences

University of Heidelberg

Master thesis

in Computer Engineering

submitted by

Stephan Proß

born in Mannheim

2023

A Hardware-Supported Boot and Debug Unit for RISC-V + eFPGA SoCs:

This Master thesis has been carried out by Stephan Proß

at the

Institute of Computer Engineering

under the supervision of

Prof. Dr. Dirk Koch

Eine hardware-untersützte Boot- und Debug-Einheit für RISC-V + eFPGA SoCs:

Externes debuggen von SoCs, die RISC-V CPUs und eingebette FPGAs kombinieren, benötigen eine Architektur, die in der Lage ist den Ausführungszustand zu kontrollieren und mit beiden Komponenten unabhängig voneinander zu kommunizieren. Auf Basis exsiterender open-source Hard- und Software untersucht diese Arbeit die Implementierung einer Debug-Architektur mit einem Kommunikationsprotokoll, das auf UART basiert. Die Eigenschaften dieser Architekur hinsichtlich der Performanz, Funktionen und Kosten werden analysiert und Design-Entscheidungen motiviert.

A Hardware-Supported Boot and Debug Unit for RISC-V + eFPGA SoCs:

External debugging of SoCs integrating RISC-V CPUs and embedded FPGAs require architecture capable of contolling the execution state and performing general data exchange with both components independently. Using existing open-source soft- and hardware components as a basis, this thesis proposes a debug architecture using a UART-based communication protocol with hard- and software to support for debugging. Analysis of the architecture regarding performance, features and resource costs is performed with the motivation behind design decisions highlighted.

Contents

1 Introduction

Whether developing traditional software or designs for programmable logic (PL), debugging and hardware introspection are crucial to ensure correct function. This is especially true for a platform for educational purposes, where the goal is not necessarily correctness of function, but understanding the hard- and software. However, even in productive environments, traceability of the architecture and observability of the systems functions are fundamental to catching errors otherwise not found in the design stage or simulation.

In the realm of traditional general purpose CPUs, traceability stems from both sufficient documentation of the system and transparency of system functions. As the system state can be sufficiently described by flags, register values and memory contents, tracing and manipulating these values constitute the basis on which debugging architecture in soft- and hardware is able to operate.

In this regard, Field Programmable Gate Arrays (FPGAs) pose a unique challenge. While the architecture of the fabric remains unchanged, explicitly implemented function changes based on user designs. This design, as opposed to the fabric itself, is the debugging target. Observation of the design's functions is difficult, as location, purpose and interpretation of signals change from one configuration to the next. Infrastructure facilitating debugging capabilities must be therefore flexible enough to fulfill most use-cases.

The FABulous System-on-a-Chip (SoC) project, currently in development and based on the FABulous project[5], has the goal of bringing FPGAs mixed with RISC-V architecture cores into the hands of educators. Using only open-source tools, the explicit goal is both traceability of the system architecture as well as tooling used

for system generation. Among the desired features is the independent operability of FPGA and the RISC-V core to allow dedicated use. Opposed to this, many commercial development boards and SoCs with similar goals are centered around one or the other. For example, boards like the Zedboard[15], featuring a Xilinx Zynq-7 FPGA[16], are core-centric and require execution of boot code to bring up the programmable logic[16]. As a related issue, pins connecting the JTAG- and UART-interfaces to the USB-converter are exclusively connected to the core and may not be used by the FPGA itself. This makes additional hardware necessary to allow designs to communicate using said interfaces through general purpose pins. This increases both cost for end-user and is inefficient considering existing connectivity cannot be used. On the other end of the spectrum, an SoC like the Virtex-II Pro [11] fully embeds the core in the PL, requiring correct configuration to be usable. Further, as the PL is also involved in the configuration, a defunct system state can occur, where loading another configuration is not possible. To prevent a bricked system, boards using this SoC feature a fallback "golden"-configuration in read-only memory.

1.1 SoC Overview

At this time of writing, the general layout of the FABulous SoC can be seen in figure 1.1. The FPGA and RISC-V cores are both connected to a system-bus based on the Wishbone specifications [**opencores**]. The system-bus itself will be realized as a crossbar, allowing independent and parallel access to peripherals by multiple bus-masters. Flash memory, acting as storage for CPU boot-code and FPGA configuration, and (pseudo-)SRAM, acting as the main memory, are external and connected via the Quad Serial Peripheral Interface (QSPI). A bridge handles translation between the system-bus the QSPI-interface. Additional interfaces connected to the system-bus include an Ethernet port, a UART-interface and a SD-Card controller. Exclusive to the embedded FPGA (eFPGA) is a double row peripheral

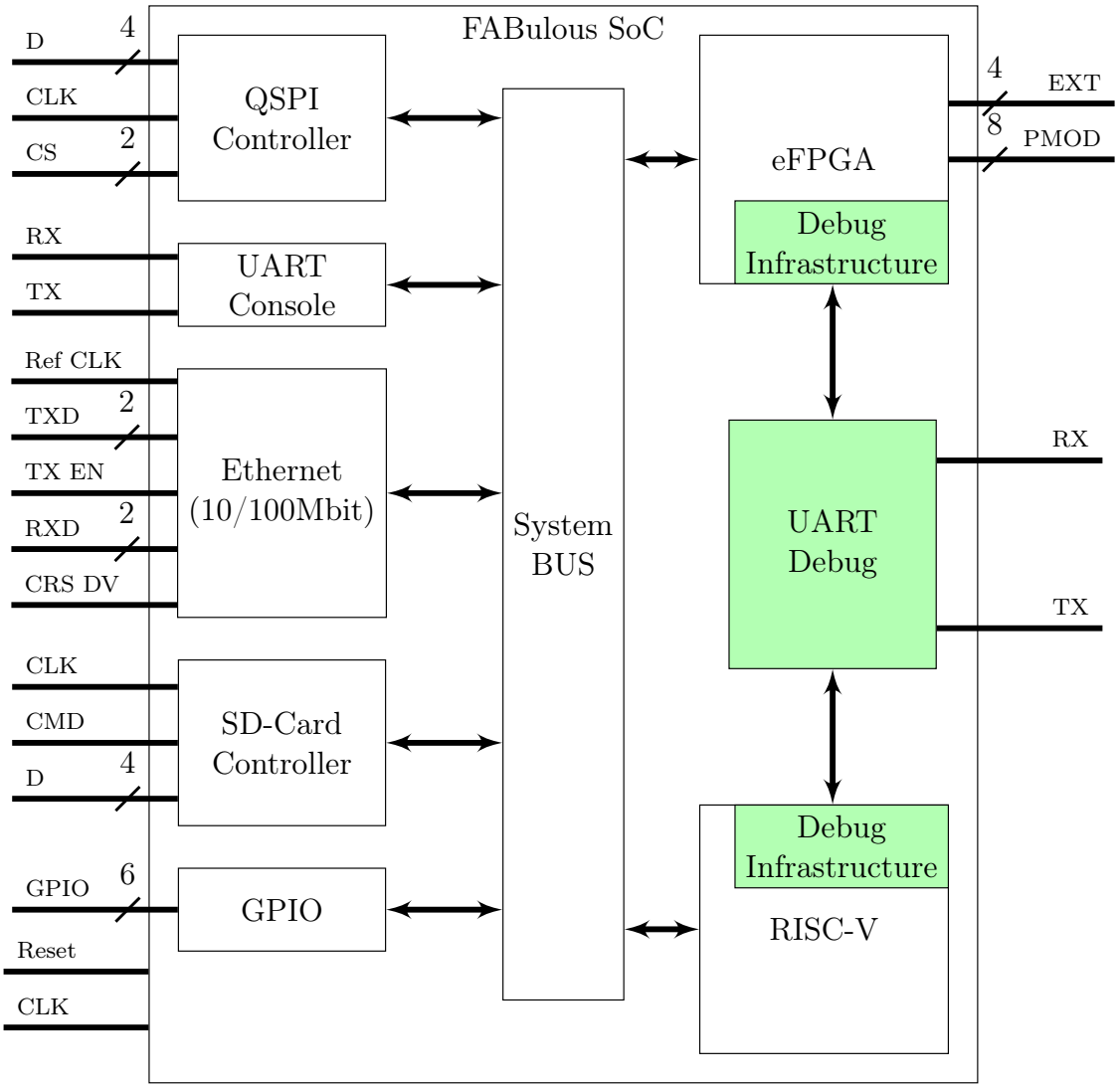


Figure 1.1: Overview of the SoC Architecture

module interface (PMOD), a small number of pins for extensions. General purpose I/O pins (GPIO) exist also and are accessible by both CPU and eFPGA through the system-bus.

The manufacturing process used will be Skywater 130nm[7] with an available area of $10mm^2$.

As the number of pins used to communicate with the outside world is limited to 44 pins, only 2 remain for the debug architecture. This limitation influenced the choice of communication interface and protocol, with the decision for UART being made.

1.2 Design Goals

The primary goals in the development of debug architecture is resource efficiency mixed with utility. From a resource perspective, the debug architecture has to fit into the available chip area. Space left over by core and system is used to implement the eFPGA. As the range of designs possible to implement in the PL is dependent on available look-up-tables (LUT) and hence size of the eFPGA, an area of 60% by the remainder of components should not be exceeded. From a utility perspective, the following features should find implementation in the debug architecture:

RISC-V Debugging Inspection and control of core state, including booting and halting the core.

Memory I/O Reading and writing to (flash) memory locations.

FPGA I/O Exchange of data between host and eFPGA through means of debug interface. The higher the throughput the wider the range of applications able to benefit from the proposed system.

FPGA Trace-Capture Capture of signals dependent on a trigger condition to allow reconstruction of events.

FPGA Control Enabling and disabling of the eFPGA fabric, reading and writing of configuration as well as possible triggering loading of said configuration from flash memory.

In the following chapters the SoC architecture will be reviewed regarding requirements to boot the RISC-V CPU and eFPGA as well as methods and standards for debugging both device classes. Design motivations and decisions regarding the proposed architecture are reviewed in Section 5. Hard- and Software details of the system with usage are provided by Section 6 and 7. Lastly, analysis of the resource costs and system performance is performed in Section 8.

2 SoC Architecture

2.1 RISC-V CPU

RISC-V is an open Instruction Set Architecture (ISA) based on principles of reduced instruction set computers (RISC). Compared to complex instruction set computers (CISC), a term only loosely defined and coined in response to RISC, RISC instructions do not mix memory operations with computations. The ISA contains multiple extensions ranging from floating point support to vector extensions. Generally, for an architecture to call itself RISC-V compliant, only the unprivileged 32bit base integer ISA extension must be implemented[12]. Precise implementation details or implementation technology are not specified and fully open to the hardware designer.

A RISC-V implementation may implement a 32-, 64- or 128-bit data path, all with 32-bit wide encoded instructions (though a compressed 16-bit ISA extension is under development). In the terminology used by the specification, a RISC-V core consists of a dedicated instruction-fetch unit and may posses multiple hardware execution threads (harts).

The FABulous SoC features a 32-bit CPU that is based on the CVA6 RISC-V project [2, 14]. This implementation of the ISA was designed with the goal supporting operating systems (OS) while remaining competitive in performance. Another goal was flexibility in the usage of the design to allow a wide range of applications ranging from embedded systems to workstations. The implementation features a 6-stage pipeline (instruction-fetch, -decode, -issue, -execute and commit stages) with optional L1-caches, floating-point unit (FPU), a memory management

unit and a scoreboard to hide memory latencies. To save resources, however, Caches and FPU will be deactivated.

2.1.1 Booting a RISC-V Core

In the general, booting involves a setup of the hardware functions and execution environments for subsequently loaded software. For example, a boot-loader may setup interfaces for memory allocation and basic I/O to a degree sufficient for allowing compiled C using the standard C-library to run. Further, a boot-loader may transition the system into a less privileged mode (from the default Machine-Mode) and activate the virtual addressing.

However, on power-on or reset, each hart of a RISC-V core executes a hard-coded sequence of operations, as summarized in the following.

Power-On/Reset

To review behavior and executed operations, one can look into the QEMU-RISC-V emulation project[8]. In total, there are 3 files which describe the behavior on reset:

- qemu/target/riscv/cpu.c
- qemu/target/riscv/cpu_bits.h
- qemu/hw/riscv/boot.c

The default reset vector is defined in *cpu_ bits.h* and equals the address "0x100", a value which may differ from one RISC-V implementation to another. A reset will cause the program counter register **PC** to be reset to this value and code at this address to be executed by each hart in parallel. The order in which the harts finish the execution of the code is not specified and can be random. It is the task of the boot-loader executed afterwards, to ensure all harts have woken up. This code at the reset vector forms a preliminary boot-loader and can be found in *boot.c*. There, the function `riscv_setup_rom_reset_vec` places the following values at this address:

Table 2.1: Perliminary Boot-Code hard-coded into RISC-V harts.

Address	Instruction	Disassembly
0x100	00000297,	auipc t0, 0
0x104	02828613,	addi a2, t0, 0
0x108	f1402573,	csrr a0, mhartid
0x10c	0202a583	ld a1, 32(t0)
0x110	0182a283	ld t0, 24(t0)
0x114	00028067,	jr t0
0x118	start_addr,	
0x11c	start_addr_hi32,	
0x120	fdt_load_addr,	
0x124	fdt_load_addr_hi32,	

The code translates to the following steps:

1. Place the value of the program counter **PC** in the register **t0**.

2. Copy **t0** over to the **a2** register.

3. Read the control and status register (**CSR**) containing the id of the hart, and store the value in register **a0**.

4. Load value from memory at address **fdt_load_addr** into **a1**. For 64-bit architectures `ld` loads a 64-bit value from memory using **ftd_load_addr** for the lower and **ftd_load_addr_hi32** for the higher 32bit of the address.

5. Load value from memory at address **start_addr** into **t0**. For 64-bit architectures, `ld` behaves as described before.

6. Jump to address found in register **t0**, equal to the start address previously loaded from memory.

In total, through the registers **a0** to **a2**, the id of the executing hart, the address containing the boot code and address of a data structure called Flattened Device Tree (FDT) are provided and form the environment of the boot-code. At this point, the processor is running still in Machine-mode, with all code executed having the highest privileges[13]. Lower privileged modes require setup by the subsequently executed code.

The boot-loader itself responsible for setting up system hardware and providing abstract interfaces for subsequently loaded code. To allow discovery of available resources on the system, the Devicetree Standard[3] exists to provide a uniform interface and data format, for communicating the available hardware.

Boot-Loader

In the RISC-V domain, there are commonly two pieces of software used as boot-loaders, both found in the RISCV-PK project[**2023b**]. The RISC-V Proxy Kernel (PK) provides an execution environment for statically linked ELF binaries. It's goal is to allow user-programs to be executed in machine-mode. To this end, functions like memory allocation, file I/O as well as console I/O are implemented and provided to user-code through system calls. The Berkeley Boot Loader (BBL) does not provide the full function of PK. Its goal is to setup memory and provide the minimal set of functions necessary to allow a Linux kernel to boot. With the kernel as payload, this involves loading the FDT to then end of the kernel memory space, setting up virtual addressing (if a memory management unit is implemented), interrupts and console I/O.

In summary, hardware supporting the boot process of a RISC-V core only needs to be able to write to the memory locations accessed by the preliminary boot code. No further features are required.

2.2 Embedded FPGA

The embedded FPGA in the SoC is generated by the FABulous FPGA framework[5]. Goal of the project is to provide a tool-chain capable of producing embedded FPGA using only open-source tools, with power, area and performance optimization being competitive to commercial FPGAs. Specific architecture, capabilities and resources of the eFPGA are yet to be decided and depend on the remaining area available after integration of CPU and the Debug Architecture.

After a reset, "booting" an FPGA involves primarily enabling the fabric after loading a configuration, referred to as bitstream, into the configuration storage of the FPGA via a dedicated port. This bitstream is generated from the users design via the FABulous FPGA framework too. The port is possibly accessible by the system-bus and potentially be capable of performing direct memory access (DMA) to read the bitstream from connected memory devices. In either case, the debugging architecture will need to be able to write the bitstream directly into the configuration port or to (flash) memory and trigger the DMA-transfer.

3 Debugging RISC-V

Implemented in every RISC-V compliant core are commands for handing over control to a debugging environments[12]. Debuggers running on the core themselves may make use of these commands. However, this requires an already running (minimal) OS on the system. Both debugging of the core after reset from the first operation as well as debugging code without a functional OS make availability of external debugging facilities necessary.

3.1 Debug Standard

RISC-V features a standard for external debugging in the current (unratified) version of 0.13.2[4]. A compliant debug module must support the following features:

Full Register Access All registers of each hart must be accessible for reads and writes by the debug module.

Full Memory Access Memory of the system must be accessible directly through the system bus and or natively through the hart's point of view.

Independent Debugging of Harts Facilities must exist to select, halt an resume execution of individual harts.

System Discovery Excluding the memory map and peripherals, a debugger must be capable of discovering everything it needs to know about the system without user configuration.

Debug at Reset Each hart must be debuggable from the first instruction executed.

Single-Step Hardware can force execution of one instructions at a time, stopping the hart after each.

Debug-Transport Independence Functionality of the debug module is independent of the type of protocol used for debug transport.

Micro-Architecture Oblivious A debugger does not need to know about the type of micro-architecture it is debugging.

In addition, the following optional features can be supported:

Instruction Execution The debugger may cause a halted hart to execute arbitrary instructions

Non-intrusive Register-Readout Registers may be read without needing to halt the hart. This can mean, that the DM has a dedicated read-port to each harts register file, or, that the registers are accessible for reading via the system bus.

Low-overhead Instruction injection A hart may be directed to execute a short sequence of instructions without halting and with little overhead.

System Bus Access Memory access by the debugger without the involvement of a hart.

Custom Halt-Triggers A hart can be halted when certain trigger conditions are fulfilled. Conditions can range from **PC** matching a certain value, access to a specific address, read or write of a specific value, or execution of certain operation. As the debug module is oblivious to the memory mapping, halt-triggers can be set to any address in the space available. Requires the implementation of a Trigger-Module.

All features require the support of so called abstract commands by the DM and special control and status registers (CSRS). Both are only accessible and addressable through external debugging infrastructure. Discovery of existing hardware and available functions by an external debugger is possible through two means:

- Presence of registers, as well as the range of available values can be discovered by writing all ones and reading back the value written. Unavailable registers or bits will return as zero.

- Supported abstract commands can be discovered by attempting to execute them. Unavailable commands will lead to the status register responsible for holding the execution state to report an error.

Dependent on the implementation of the required features, some of the optional ones can find fulfillment as a byproduct, while other specifications can become irrelevant due to system architecture, i.e. a single core system. Given the goals set at the beginning, specifically the independent system-bus access is a desired feature as this would allow writing to connected memory without involving a hart. However, specific implementation of this feature can differ.

3.1.1 Standard Architecture

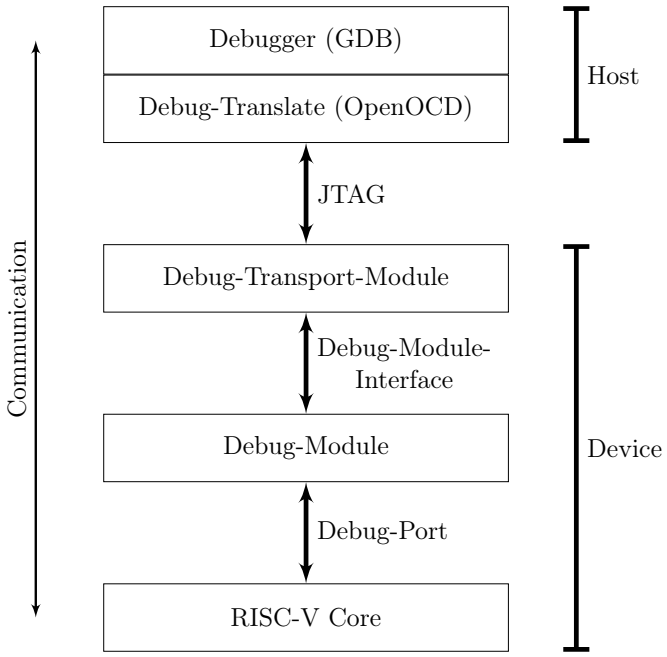


Figure 3.1: Debug Architecture as defined by the RISC-V Debug Standard.

A the debug infrastructure based on the standard consists fo 3 distinct modules:

Debug Transport Module (DTM) Handles communication with the debug host system. Typically implements a JTAG Test Access Point (TAP) and translates requests to abstract commands sent over the supported interface.

Debug Module (DM) Converts abstract commands into control signals for connected harts. Dependent on the implementation, a DM may also feature a program buffer and or access to the system bus.

Debug Port Aggregate of all control signals to allow debugging. Must give direct access to the general purpose registers (GPR) or at least allow functionality indirectly through arbitrary code execution.

Debug Module

From the specifications stated before, the following requirements on the actual DM arise: A DM must:

1. provide necessary implementation information,
2. allow individual harts to be halted and resumed,
3. be able to discover, whether a hart is halted or not,
4. be able to access to general purpose registers (GPRs) of halted harts,
5. be able to debug a hart from the first instruction executed.

A DM must also implement at least one of:

1. A Program Buffer for executing arbitrary instructions by a chosen hart.
2. Direct System Bus Access
3. Memory access from the hart's point of view.

Optionally, a DM may feature:

1. Debugging of a hart immediately out of reset.

2. Ability to halt, resume or reset multiple harts simultaneously.

3. Abstract access to non-GPR hart registers.

A DM may implement all of the features above but has to at least allow the usage of a Program Buffer or access to all registers visible to a program running on the hart. For implementing the above functions, the DM contains its own set of registers only accessible by the DMI. Of specific interest is the **command** register with connected abstract data registers **data0** to **data11**, used for passing parameters and return values. These allow the following set of abstract commands:

Access Register Allows access to all registers of a selected hart. Register address is encoded in the abstract command itself with contents being either read from or written to the abstract data registers. Automatic increment of the register address can be enabled and automatic execution of code contained in the Program Buffer can be triggered.

Quick Access Halts a selected hart and causes it to execute code contained in the Program Buffer and resume afterwards. As the Program Buffer itself is optional, so is this command.

Access Memory Using the values contained in the abstract data registers as an address, either read from memory to the abstract data registers or write in the other direction. Which data registers contain address and which the value depend on the data width of the architecture. Access happens from the point of view of the selected hard in Machine-mode.

With these commands, access to the system bus can be realized through different means. The abstract command for memory access can be used for this purpose, as can the Program Buffer by forcing a hart to perform the access. However a DM may also implement an additional set of registers to facilitate communication and control between DMI and a system bus avoiding any interaction with a hart.

3.2 Existing Implementations

3.2.1 VexRisc

The VexRisc project[10] contains a debug plugin which implements the above functionalities by being able to halt the execution pipeline of the RISC-V core and force the execution of its own instructions. The plugin supports commands for flow control, command injection, as well as setting and reading hardware breakpoints for the program counter. Despite support by open-source debug-translation software, which allows full debugging capabilities of a connected debugger, the VexRisc debug plugin does not support the RISC-V Debug Specification.

The debug module does not feature infrastructure for the automatic discovery of existing hardware configuration. Instead, YAML files containing the system configuration are created on compilation of the VexRisc core and are subsequently read out by OpenOCD. In addition, the encoding of the abstract commands does not conform to the standards described in the specification, likely to avoid translation overhead between the System Debugger and Debug Plugin.

3.2.2 RISCV-Debug

The debug support of CVA6 is realized by the RISCV-Debug project[9] which is compliant to the RISC-V Debug Support standard version 0.13.1. While execution based, the same as the VexRisc debug plugin, RISCV-Debug also utilizes a Program Buffer of 16 words in addition to abstract commands. Additionally, the following features are supported:

System Bus Access (SBA) Access to the system bus as a master is possible, independent of any hart. The protocol implemented in the SBA is AXI as used as a system bus in CVA6.

Program Buffer as Bus-Slave The memory interface used as the Program Buffer is connected the system bus as slave device, allowing a hart to read its contents.

Full Register Access In addition to access to general purpose registers (GPR) of each hart, control signal registers (CSR) can be read and written as well.

DTM with JTAG JTAG is used for communication with a host system.

4 Debugging FPGAs

Commercial SoCs like the Zynq-7000 series [15] provide debugging infrastructure in the form of trace-acquisition logic, trigger logic and means of exchanging basic data.

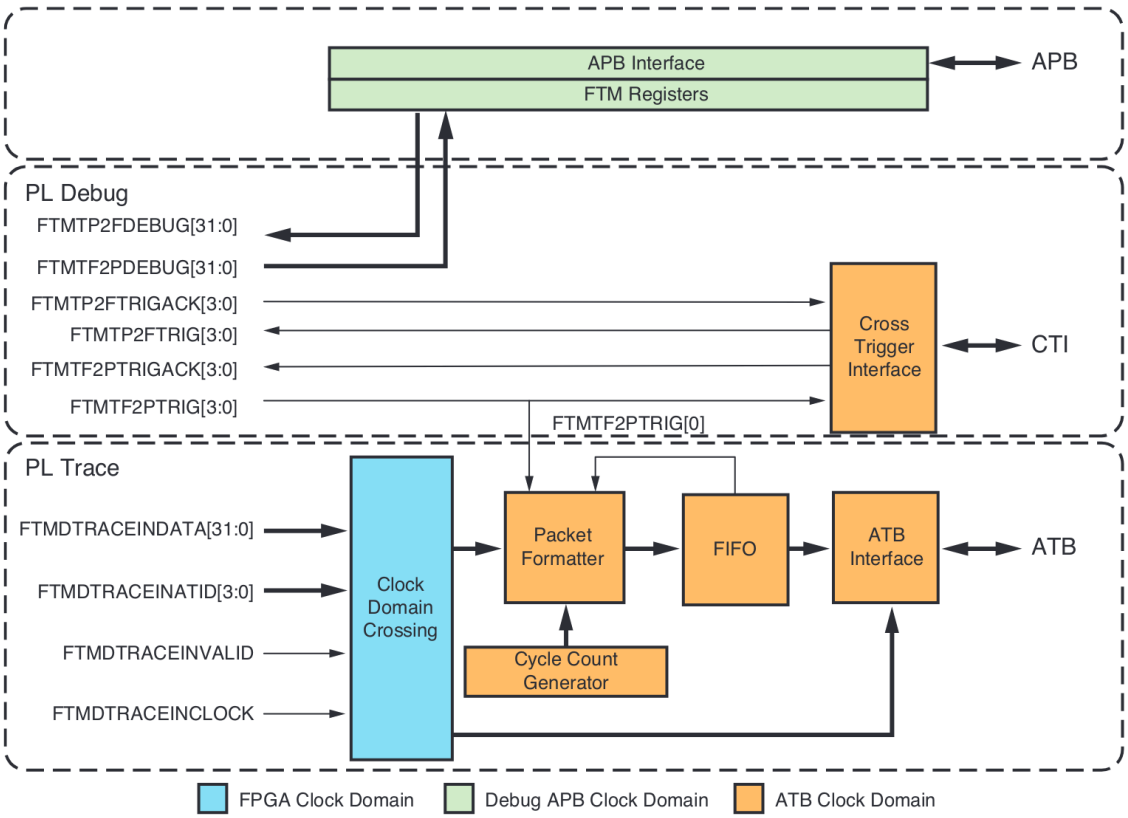


Figure 4.1: Debug infrastructure found in Zynq-7000 series SoCs[15].

Figure 4.1 shows how these functions are implemented in the Zynq-7 SoC. The trace-acquisition logic captures traces over the clock-domain-crossing (CDC) between FPGA and debugging infrastructure. Further logic generates packets from the captured traces, buffers these in a FIFO and to allows exchange with the debug

communication facilities. The signals traced, together with the clock of the CDC, must be provided through routing in the users design. Similarly, the trigger logic found in the Zynq-7000 SoC is implemented, by the most part, through the users design as well. A form of basic data exchange is possible through two 32-bit registers between the system and PL using the Advanced Peripheral Bus (APB) interface.

In order for a host to communicate with the supplied logic, a common communication interface for commercial systems is the JTAG (Joint-Test-Action-Group). JTAG requires the usage of a minimum of 4 wires to carry the signals named **TMS**, **TCK**, **TDI** and **TDO**. Communication partners share the same clock provided by the host via **TCK**. Data is exchanged serially via **TDI** and **TDO** in a synchronous and daisy-chained fashion. To differentiate between data and command, **TMS** carries a signal controlling a state-machine internal to the TAP of the device. Multiple TAPs may be chained together, with all TAPs sharing the same **TMS** signal and each TAP **TDO** connected to the **TDI** of the TAP next in chain.

5 Architecture Design

With the goals defined in chapter 1.2, to avoid redesign of already functional architecture, reuse of as many tools as sensibly possible was an implicit goal. In terms of hardware design, this meant that the already capable debug infrastructure for the CVA6 core provided by the RISCV-Debug project would form the basis of the implementation. From the standpoint of software, support for the GNU-Debugger[6] (GDB) as well as the defacto standard of debug-translation OpenOCD[**zotero-243**] in the RISC-V world was a declared goal.

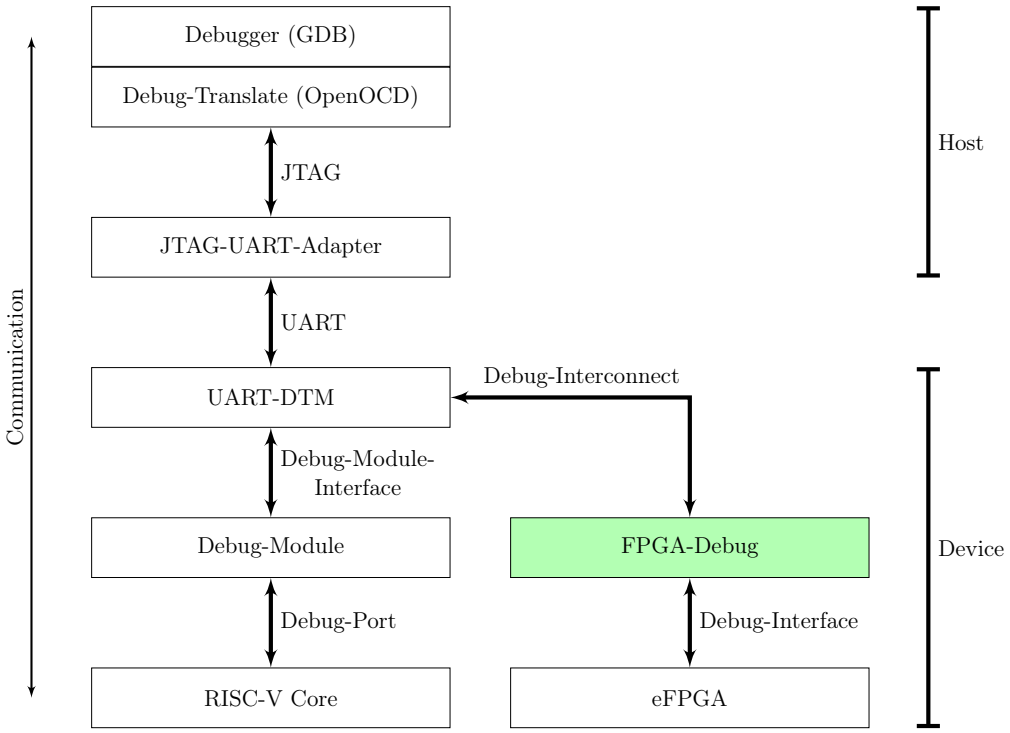


Figure 5.1: The Proposed Debug Architecture.

Compared with the original debug stack in Figure 3.1, the use of the UART-Interace was reason for many of the necessary changes. The following components required implementation:

JTAG-UART-Adapter As OpenOCD does not support a UART as a transport protocol, a custom adapter program translating JTAG commands into the communication protocol used by the DTM was necessary. This solution was preferred over extending OpenOCD by the UART-Interace as the abstraction provided by this adapter ensures compatibility with future versions.

UART-DTM To make use of the properties of the UART-Interace and to interpret instructions sent by the Adapter, a custom DTM had to be implemented. Additionally, a fast data-path to the FPGA through the debug architecture made allowing direct access through the DTM necessary.

FPGA Debug Architecture Trace-Capture, Trigger handling and data exchange between Host and FPGA find implementation in the FPGA Debug Architecture. Additionally, as the FPGA and System lie in different clock domains, the CDC needed to be handled by the system.

5.1 Communication Protocol

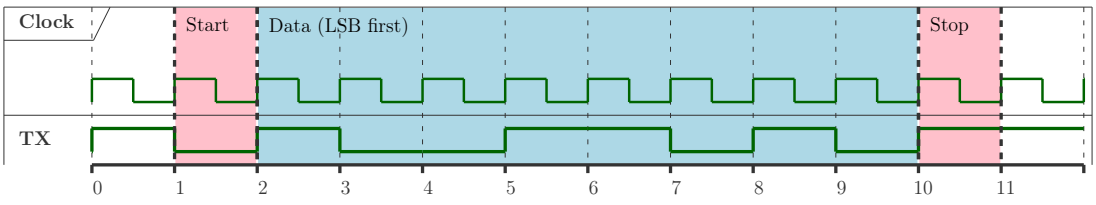


Figure 5.2: Transmission of a UART-Frame

UART (Universal Asynchronous Receiver Transmitter) is a communication standard uses only two wires, **RX** and **TX**. As a common clock signal is omitted, transmission between partners must adhere to a common symbol rate, the baud

rate. For UART this is equal to a bit. Data is transmitted in frames consisting of a start bit, the data itself (5-9 bits), an optional parity bit and a stop bit. The most common setting consists of 8 data bits without parity and one stop bit. As a result, 80% of the communication contains usable data. Therefore, the effective data-rate of UART is 80% the baud rate.

As the UART-Interace only transmits data, an additional communication layer in the form of a debug protocol had to be developed to differentiate between instructions and data.

5.1.1 JTAG Over UART

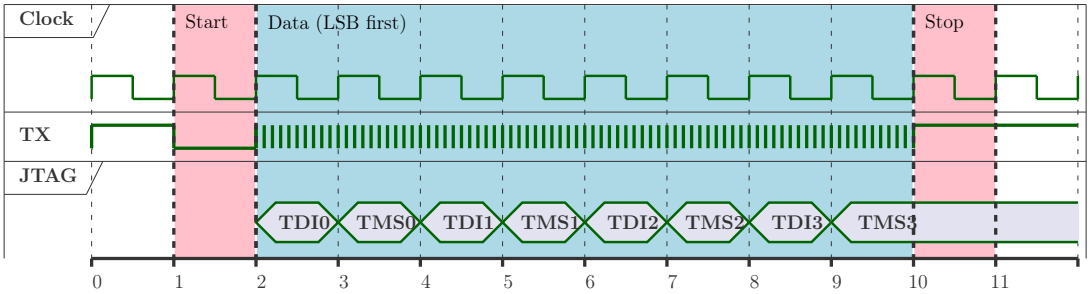


Figure 5.3: JTAG signal wrapped in a UART-Frame

Wrapping the common JTAG interface for debugging into the UART communication allows the reuse of existing hardware. DTMs for JTAG can be reused by adding an hardware-wrapper unpacking the UART-frames, while on the side of the host, a software wrapper for JTAG to UART allows reuse of OpenOCD.

On the other hand, wrapping JTAG into UART poses the following challenges:

TMS and TDI Since JTAG features two input signals, both need encapsulation in pairs into the UART-frame. Considering only the data over **TDI** as the payload, the effective data rate is halved using this simple scheme.

TCK The state machine of a TAP-Controller is dependent on the value of **TMS** as well as the clock signal provided by **TCK**. Since UART is asynchronous,

a new clock signal needs to be generated for each received pair of **TMS** and **TDI**.

JTAG-State-Machine Assuming that previous communication had shifted data into the instruction register of the TAP-Controller, in order to transmit data into a data register, 4 cycles of **TMS** signals are required. With the previously mentioned pairwise wrapping of **TMS** and **TDI** signals into a UART-frame, this switch would need one frame of only **TMS** signals before data can be transmitted.

TDO Rate Matching Given that JTAG is synchronous, the data-frame sent back to the host in response cannot contain more data bits than previously sent.

In the best case scenario, i.e. writing data to a register after bringing the state-machine into the correct state, the effective data rate is reduced to only 40% of the baud rate of the UART-interface, as only four of 10 bits of the UART-frame carry data. It is for these reasons, that this protocol was not chosen.

5.1.2 Instruction Bit

Implementing a bridge using purely the available resources of UART comes with its own challenges:

Interface Specifications The RISC-V Debug Support Standard 0.13.2[4] defines registers with connected functions (**IDCODE**, **DTMCS**, and **DMI**) normally implemented by the JTAG interface. To keep compatibility with existing tools, similar registers have to be implemented by the UART-DTM.

Address vs. Data To access these registers, data transmitted over UART must feature an indicator to differentiate between instructions selecting registers and transmitting data.

As a solution,a single bit to of a UART-Frame may reserved, to allow differentiation between data and instruction.

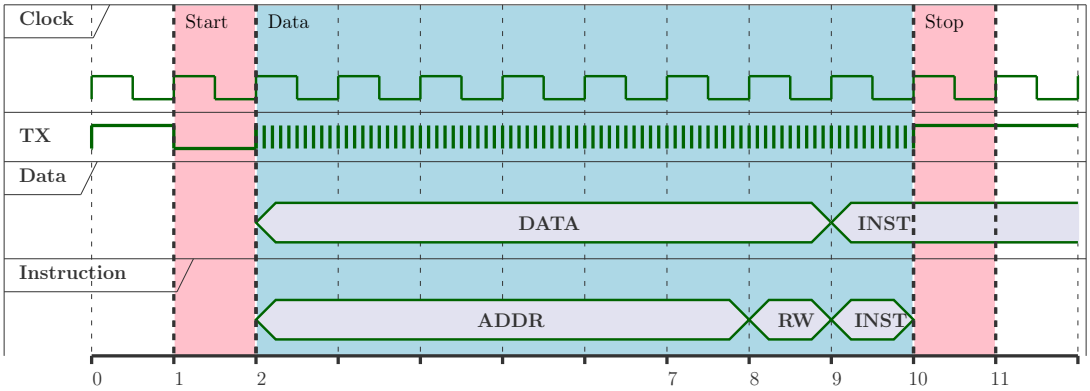


Figure 5.4: Encoding of instruction and data using one bit to differentiate. Instruction frame consists of address as well as a bit to indicate reading or writing intend of the operation.

Considering a full 32-bit register as target, 5 transmissions of 7-bit data frames are needed for a full write. This leaves 3 bits of the last data frame unused if no alignment with subsequent transmissions to the same register is practiced. Ignoring alignment issues, this protocol would reduce the effective data-rate to only 70% the baud rate in the best-case.

5.1.3 Instruction Packets

Another option involves arranging multiple frames into a packet consisting of a header, an instruction combining command and address and number of data bytes contained in the packet.

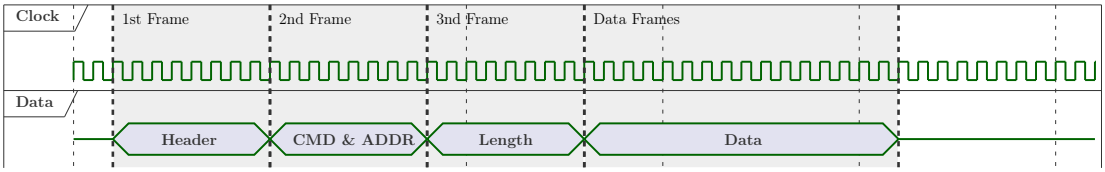


Figure 5.5: Data packet consisting of header-, instruction-byte, size and data payload.

The RISCV-Debug Standard defines a variable address length for accessing registers in the TAP. The minimum number of bits required is five, leaving three to encode arbitrary commands. Used during the first implementation variants due to

its simplicity, this packet-based protocol variant was retired due to the imposed overhead: While the number of data bytes may be omitted, so long as both sender and receiver are aware of the lengths of addressed registers, each transaction involves at least an overhead of two bytes. The resulting effective data rate depends on the number of bytes transmitted and therefore the length of the addressed register. Assuming a default register length of 32bit, the effective data rate drops to approximately 53%.

5.1.4 Instruction Escaping

Instead of reserving a single bit of the UART-frame, an escape character in the form of a single byte is reserved for differentiating between instruction and data. A frame transmitted directly afterwards is interpreted as an instruction consisting of command and address, similar to the format described in the packet-based protocol. To allow the this character to be interpreted as a data by the receiver, the character has to be sent twice. As such, a combination of command and address equal to the escape character must be avoided through protocol definition. Due to missing any information in the instruction regarding data lengths, both device and host must be informed of addressed register lengths by definition. However, if the register addressed remains target of subsequent write operations without needing another instruction, continuous writes or data streaming is possible without additional overhead. Assuming the case of random data sent by the host to device with linear distribution, this protocol would reduce the effective data-rate to only approximately 79.7% of the baud rate. However, such a scenario is theoretical, as communication involves patterns, causing some characters to be more likely than others. In the worst-case scenario of all data transmitted being equal to the escape-character, the effective data-rate would drop to 40%.

While optimal choice if the escape character requires further analysis of the expected communication, this protocol has been chosen and implemented in the DTM and Adapter program.

5.2 Clock Domain Crossing

Due to the different clock domains of FPGA and system, peripherals designed to interact with both need to take special care handling signals crossing those domains. While the FPGA domain is guaranteed to be equal to or slower than the system clock domain, signals crossing from the faster to the slower domain can exhibit metastability. This is commonly the case, when a signal flips polarity too close to a clock edge of a sampling Flip-Flop (FF), violating setup & hold times.

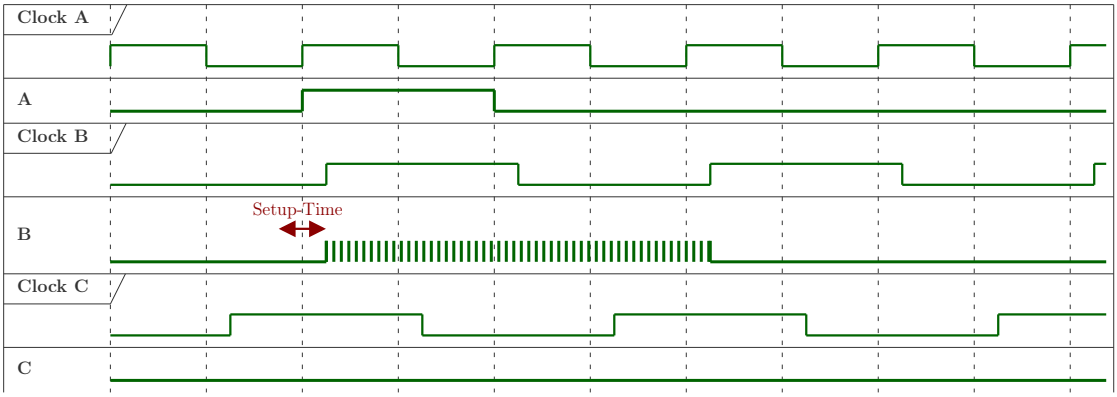


Figure 5.6: Signals B and C sampling A. Signal B metastable as the result of setup-time violation. Positive edges of clock C causes sampling to miss change in A.

Sampling such a metastable signal will lead to an unknown state by subsequent logic and should be avoided. Techniques exist to synchronize signals between clock domains, with some of the commonly used methods explained below [1]. In general, all synchronization techniques trade signal delay for a higher accuracy of signal acquisition. Further, if all values need to be sampled by the receiving domain, synchronization can also have a large influence on throughput as the sender has to wait on synchronized feedback before allowed to proceed.

5.2.1 Synchronization Techniques

As a general requirement, both sender and receiver must register their signals in their respective domain before communication can occur. This removes possible

delays imposed by combinational logic and secures signal stability for at least one cycle in the domain of the sender. This, however, is not sufficient to guarantee avoidance of metastability as ratio and phase relation of the clocks can still produce a situation as in Figure 5.6. With this requirement, the most resource efficient technique, the open-loop (OL) synchronizer, adds another FF to the signal path in the receiver domain. This increases the delay of the signal acquisition by one cycle of the receiver clock, with the time being used to allow the metastability of first FF to settle. Subsequent sampling by the second FF will lead to a valid signal [1].

However, synchronization may still fail under the condition, that the sending signal changes 1.5 times faster relative to the receiving domain. If this condition is violated, a situation may occur where the first FF becomes metastable again before the second has a chance to sample a stable signal. As a result, since the FPGA may be as fast as the system clock domain, signals from system to FPGA must be stable for at least 2 cycles in the system domain to ensure synchronization. If frequency ratios between the two domains were fully unknown, a closed-loop solution may be formulated by feeding the synchronized signal from the receiving domain back to the sender. This signal, again synchronized by two FFs, can be used to ascertain correctness of the transmission. However, this solution not only involves more FFs, but also increases signal delay compared to the OL synchronizer.

If multiple signals need to be synchronized, OL-Synchronizers are insufficient to ensure all signals arrive at the same clock cycle in the receiving domain. Small differences in trace length or other environment factors can skew arrival time of signals. While OL-Synchronizers guarantee eventual correctness of each signal, the correct cycle to sample these values is unknown to the receiver. This problem is solvable by using an additional synchronizing signal, which marks the cycle in which the data received is valid. This formulation, called Multi-Cycle Path (MCP) formulation, changes the logic compared to OL-Synchronizers in the following way: As synchronization is done via a pulsed load signal, the actual data must only be registered once on each side using gated D-FFs. A load-pulse on the sender side

initiates transmission and, using additional logic, is turned into a toggle-signal, synchronized on the receiver-side and lastly turned into a now synchronized load pulse again. This load-pulse in the receiver domain forms the synchronization event.

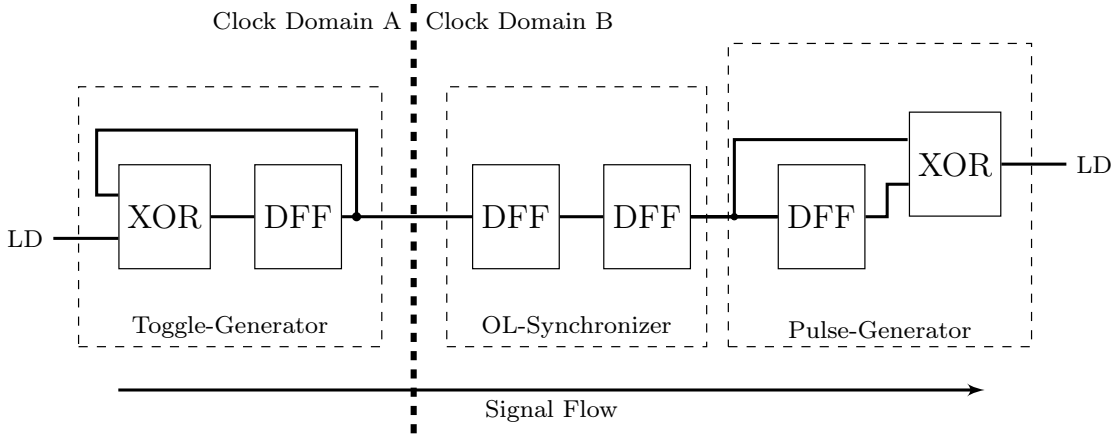


Figure 5.7: Synchronizing signal of Multi-Cycle Path formulation.

The delays introduced by the MCP-Formulation ensure signal stability at the inputs of the gated D-FFs, while the synchronized load pulse denotes the exact cycle for which the data is valid. The following criterion apply to correctness of the MCP-Formulation:

- As an OL-Synchronizer is part of the load-toggle synchronization process, the load-toggle must be stable for at least 1.5 times the receiver clock. As the FPGA clock domain is slower or equal to the system domain in our case, this condition is satisfied through the requirement of a pulsed load.

- Data on the receiving end is only valid during and between the synchronized load-pulses.

- Unsynchronized data must be stable during the sending process. The sender may not start a new transmission until the synchronized load pulse has occurred on the receiving end. Violation of this condition may inflict metastability.

To ensure the last criterion, the time between load pulses must be equal to or larger than the synchronization delay of the receiver. With known clock ratios beforehand, this would be possible to hard-code into the design. As the FPGA is arbitrarily slower than the system, a form of synchronized feedback from receiver to sender is necessary.

Synchronizing multiple signals at once is also possible by using asynchronous FIFO with write port in the sending and read port in the receiving clock domain. Pointers need to be synchronized between domains to detect full or empty state of the FIFO. Without this prevention, reading may yield old or undefined values if empty, while writes displace values not yet read by the other side if full. To simplify the synchronization of pointers between the domains, the pointers themselves are usually Gray-Encoded. As a result, each pointer increment causes only one bit to flip which allows the usage of OL-Synchronizers as opposed to more expensive MCP-formulations.

5.2.2 Crossing Design

Other than the aforementioned synchronization techniques, perhaps a more important decision is the design of the crossing and necessary communication. If part of the logic can be freely placed in either clock domain, the choice of partitioning influences the number of signals synchronized as well as the choice of synchronization technique. In turn, delays imposed by synchronization need to be considered by the logic and pose a trade-off. In total, the following properties need to be considered:

Signal Change Rate Signals synchronized must be stable for at least 1.5 times the rate of the sampling clock domain. OL-Synchronizers are no longer reliable below that threshold. Alternative partitioning and redesign of the logic has to be considered.

Signal Count Reducing the number of crossing signals saves resources spent on synchronization. Further, for a chosen partition, signals can be consolidated,

if they are derivable from other synchronized sources using a mix of combinational logic and delaying FFs.

Sampling Feedback If each signal change needs to be sampled by the receiving domain, a synchronized feedback signal from receiver to sender needs to be added to the crossing.

Signal Grouping Synchronizing multiple bits using the MCP-formulation requires analysis of independence of synchronization events. Two MCP-Synchronizers may be replaced by a single one, if data of the first is only processed in response to the synchronization event of the second.

6 Hardware Implementation

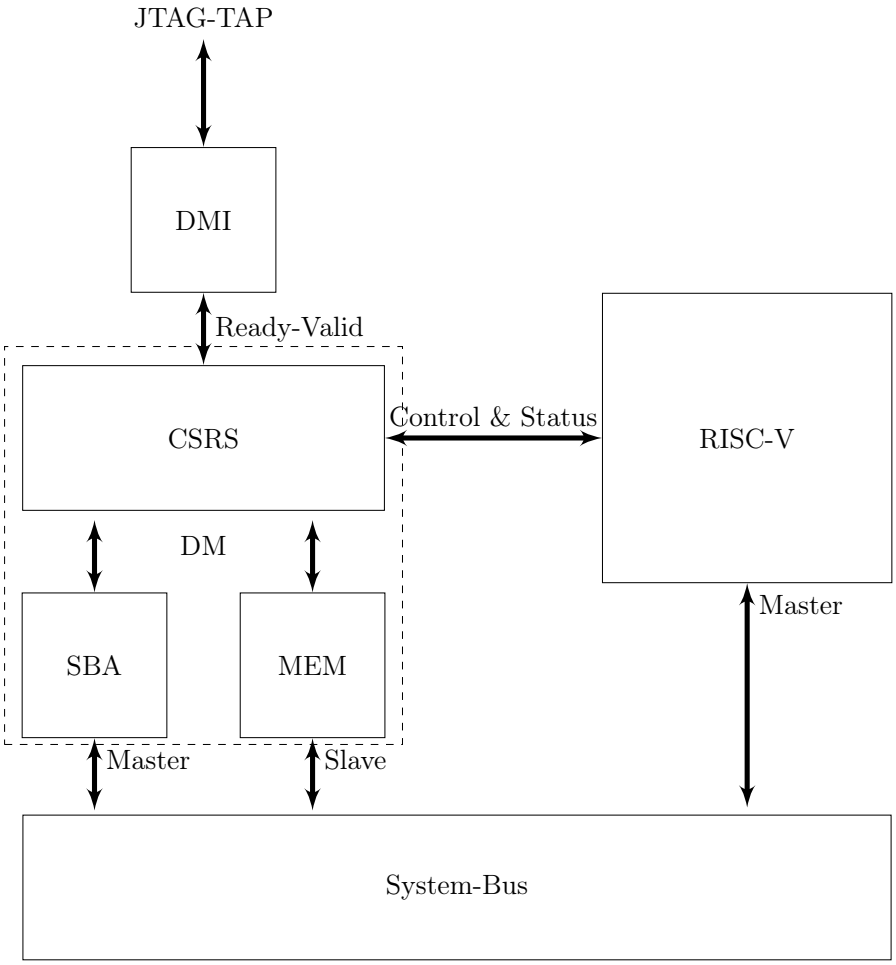


Figure 6.1: Existing RISCV-Debug Architecture.

Basis for the hardware implementation was the existing DM by the RISCV-Debug project. This module is capable of accessing the system bus, both as masters and slave, without involving any hart. Control over the execution state of the harts is carried out by halt- and resume-request signals for each individual hart, with

feedback signals allowing querying the execution state. Out of box, the DM is able to debug a core consisting of multiple harts. Figure 6.1 shows the constituent components of the DM:

DMI The DMI represents the point of connection between the DTM and the DM. In the original implementation, the DMI module contains the CDC between JTAG and system domain. Primary purpose is handling the access of the DMI register, which has a slightly different ordering of fields compared to the OpenOCD definition. As the original DMI implemented a second JTAG-State-Machine parallel to the TAP, this module had to be rewritten to function using the UART-DTM.

CSRS Directly connects to the DMI and contains and handles DM registers. Is connected to every hart of the CPU and is able to exert control over harts using dedicated signals. Registers facilitating system bus access and Program Buffer are forwarded to and from the MEM and SBA modules.

MEM Holds both the Program Buffer, writeable by the host and DMI, as well as a small piece of memory for abstract command execution, the abstract command buffer. The module is connected to the system bus as a slave device to allow harts to read, write and execute the Program Buffer and execute the read-only abstract command buffer. A halted hart executes code at a specific address corresponding to the abstract command buffer.

SBA Connected to the system bus as a bus master, allows reading and writing to addresses defined via DM-Registers. Unlike access via abstract commands, accessing memory using the SBA involves no hart.

Future changes to the DM may include removal of the Program Buffer to reduce resources required by the module.

6.1 UART Debug Transport Module

In order to interpret packets sent in the debug protocol defined in section 5.1.4, a new debug transport module had to be implemented. Aside from functional correctness of the communication between host and device, utilization of asynchronous nature of UART and available data rate were explicit design goals.

6.1.1 UART-Interface

Despite the simplicity of UART, implementing a functional interface posed several challenges. While the transmission rate and hence sampling frequency must be predetermined between communication partners, there is no common clock, allowing for arbitrarily shifted signal phases. This issue is solved, by tying the signal sampling to the falling edge of the start-bit. In addition, by sampling data bits in the phase center, small deviation between phases are tolerated. In order to divide the system clock down to the baud rate, a counter using the following maximum value C given by the following equation is used:

$$C := \lfloor f_s/b \rfloor \tag{6.1}$$

with f_s being the system clock frequency and b being the baud rate. Rounding of the sampling rate f_s stems from integer division and poses problems for certain ratios of clock frequency to baud rate.

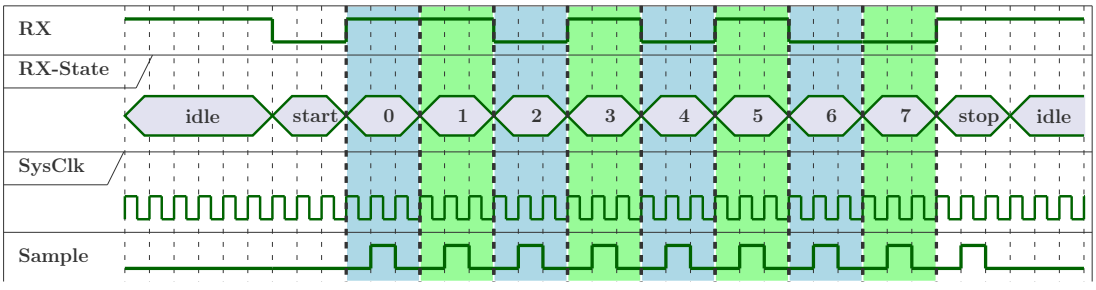


Figure 6.2: Sampling of data bits for baud rate of 3MBd and system clock frequency of 10MHz.

As 6.2 demonstrates, the actual sampling rate does not match the baud rate to such an extend, that accumulated deviations lead to incorrect assessment of data bits. In this example, the receiver would be able to detect this mistake as sampling the stop bit would return an unexpected zero. However, data would not be recoverable, causing loss of the entire frame.

To solve this issue, two solutions have been considered:

Phase correction using edge detection

Similar to the falling edge detection to start the sampling process, edges in the data bits can be used to correct phase of the sampling. This necessitates logic, which keeps track of the expected and observed edges in the signal, with the difference used to correct the sampling interval. This solution has drawbacks:

- Resources: At least two more adders are required. One for the difference between expected and observed edge and one for the correction of the sampling interval.
- Data-dependence: Frames must carry data bits shifting polarity at a higher rate than the deviation can accumulate to cause incorrect sampling. A frame containing only zeros as data would yield the same error as found in 6.2 as no edges exist for correction.

Remainder counting

As the deviations stem from integer division, taking the remainder of the ratio between clock and baud rate into consideration helps alleviate this issues. With values from previously, the floor operation removes exactly 1/3 from the result. To correct and prevent accumulation of phase deviations, it would be sufficient to delay every third sample by one cycle. More generalized, a sample-counter delaying the

sample by one cycle on overflow, counting to R given as

$$R = \lfloor \frac{(f_s \mod b)}{b} * 10 \rfloor \tag{6.2}$$

is able to correct and prevent accumulation of error. The decimal points of the remainder beyond the first may be ignored as deviations introduced by these are unable to accumulate to a full clock cycle over the duration of a single frame. Unlike the first solution, remainder counting is independent of the data contained in the UART-Frames. Additionally, as the counter is small, this solution is also to expected smaller and therefore preferred.

When debugging designs for FPGAs, simple communication with the outside world in any form can be helpful to determine proper function. Due to the simplicity of the UART protocol, a frequency divider and a shift register containing a predefined frame are sufficient to facilitate basic signaling. Normally, to allow such communication, dedicated **RX** and **TX** pins would have to be provisioned for this purpose in the SoC-Design. Given that the number pins are a scarce resource, communication channels have been added to the UART interface based on the following observation:

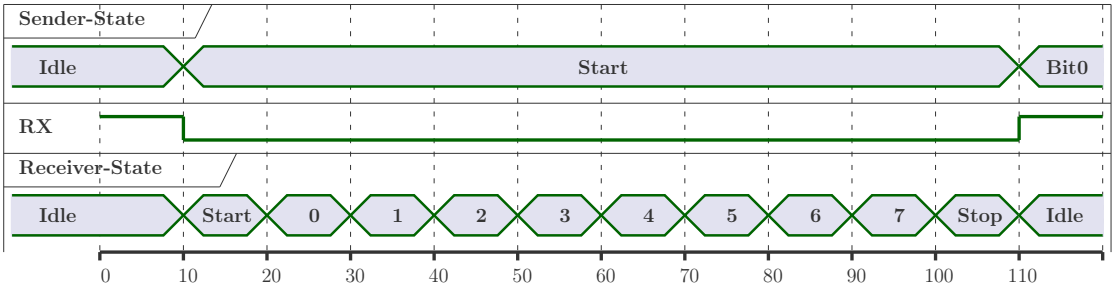


Figure 6.3: Sampling of a signal with a baud rate 10 times slower.

Figure 6.3 shows the difference in interpretation of a frame by the receiving interface against the intended meaning by the sending interface. With the sender being at least 10 times slower than the receiver, the latter moves through its de-serialization

process within one bit of the sending frame. As the stop bit will always evaluate as zero, the frame will be considered faulty by the receiving interface.

This behavior can be utilized to automatically switch the target interface on the reception of a slow signal, without additional physical connections or protocol layers. Due to the higher bandwidth requirements and existing logic, the fast interface is connected to the TAP, while the slow to the FPGA. If a slow signal is detected by the interface, both **RX** and **TX** communication is forwarded to the FPGA, while still being monitored for fast signals. Conversely, a fast signal will cause switching the **RX** and **TX** back to the TAP. Handover to the slower interface may also bet triggered by the TAP itself. In either case, switching from the slower to the faster device will cause the former to briefly assert a falling edge in the RX line. To avoid invalid detection of a frame by the slower participant, the **RX** line is pulled to zero, causing a de-serialized frame to be invalid due to zero stop bit. This can also be used by the user's design to detect missing read & write privileges.

With a system clock of 25MHz and maximum supported baud rate of 3 MBd, the implemented interface will have more than 80 cycles to process incoming frames. Despite this, situations may occur where the destination of the data is busy or otherwise incapable of processing. In such an event, where a frame hasn't been processed before the arrival of another, data loss is unavoidable. Due to much greater execution speed and available buffers of a debug host system, data sent from the interface to the host is of no real danger of being lost. The problem persists however in the other direction. Despite large resource costs, a small FIFO with 8 bit words and a depth of four was inserted between the RX module.

Figure 6.4 shows the resulting architecture. In the event, that the FIFO is half-full, the **TX** module will circumvent incoming data and send the escape-sequence to the host at the earliest opportunity. In the worst-case scenario, when the FIFO is exactly half-full as the **TX** module has begun a transmission, the delay between this event and the arrival of the escape sequence allows two full frames to be received in the meantime. As a result, the FIFO must be able to buffer at least two frames at

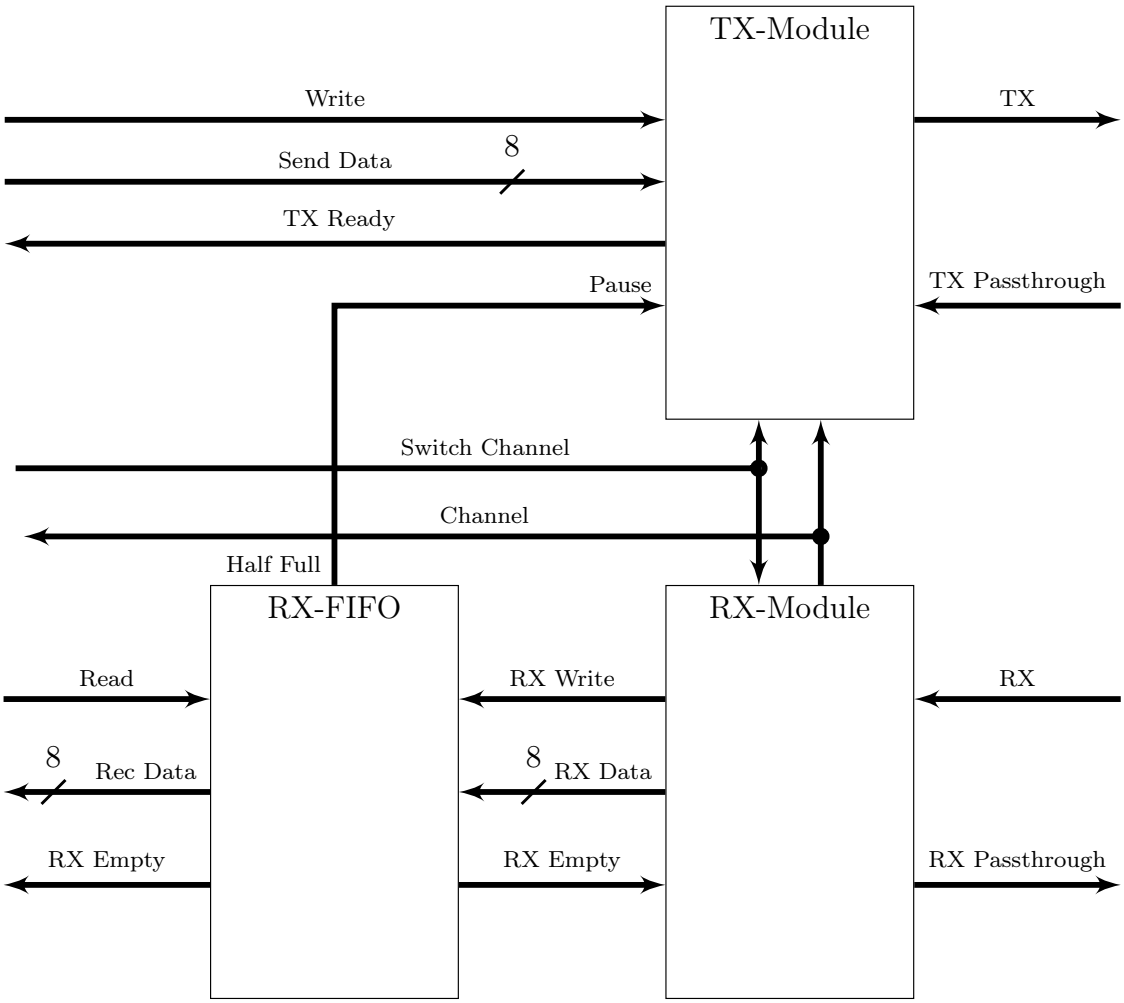


Figure 6.4: UART-Interface

all times to allow pausing the write stream by the host. However, a FIFO with only a depth two would necessitate sending the pause sequence each time a new frame has been received. To prevent continuous pausing and restarting the write-stream, the FIFO must be larger than two. The depth of four has been chosen, as sending the sequence to restart the stream also requires time potentially leaving the TAP idle.

6.1.2 Test-Access-Point

Due to the changes imposed by the usage of the UART as opposed to JTAG, the original Test-Access-Point (TAP) of the RISCV-Debug Project[9] had to be replaced by a new implementation. The UART-TAP implemented has the following features.

RISC-V Debug Registers The registers **IDCODE**, **DTMCS** and **DMI** exist and are accessible by the same addresses.

Byte (De)-Serialization Bytes received from the UART-Interface are automatically de-serialized to the correct register sizes, with registers in the other direction serialized into bytes.

Write-Streaming Once a write address using a write command is set, subsequent writes will be directed to the same address.

Host Notification Peripherals connected to the TAP can signal availability, which automatically notifies the host.

Read-Streaming Registers representing access to peripherals can be read repeatedly using one command, so long as peripheral signals availability.

Write-Cancelling At any time, de-serialization can be cancelled side-effect free by issuing a write-command to a zero address.

Peripheral Access TAP must be able to perform read and writes to connected peripherals.

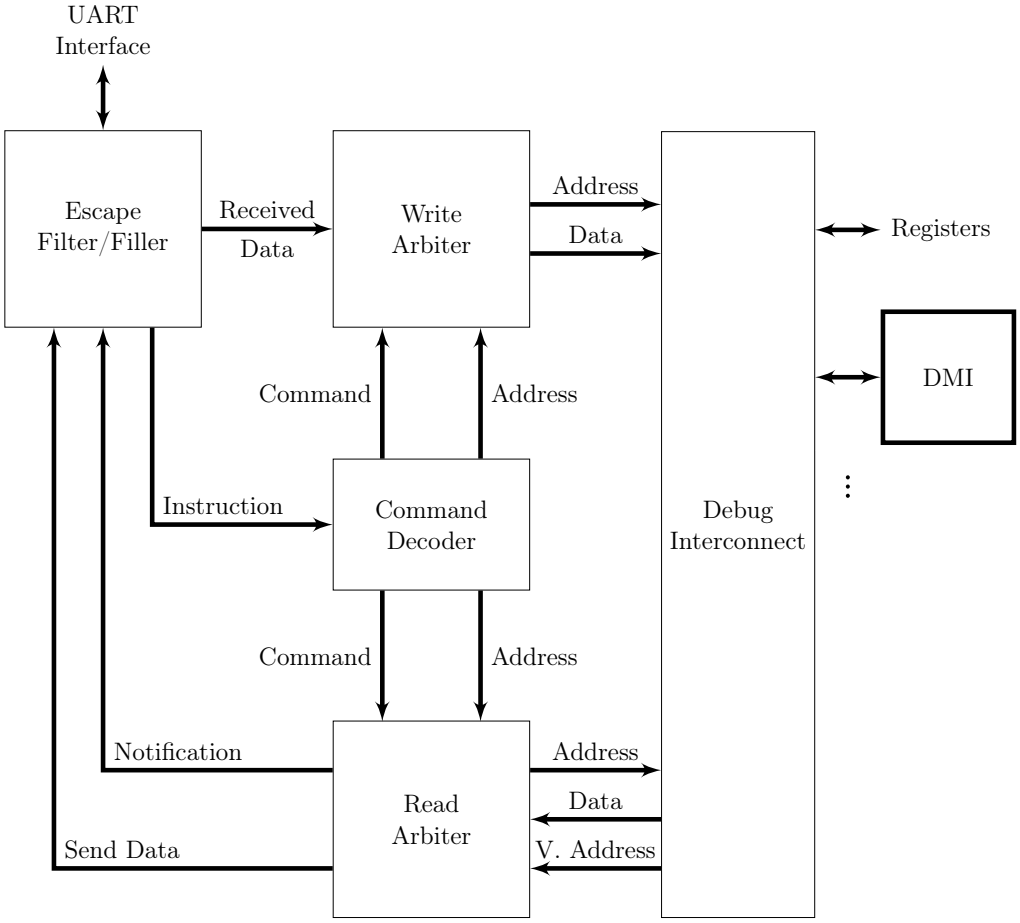


Figure 6.5: Test-Access-Point Architecture. Not shown are ready-valid signals between components.

From the UART-TAP consists of the following modules.

Command-Filter/Filler Scans incoming and outgoing bytes for the escape character. An incoming escape sequence with instruction is directed to the Decoder instead of the Write-Arbiter. An outgoing escape character is repeated to prevent interpretation of the following word as instruction. The module will prepend the escape character to the instruction received by the Decoder. Forwards TX-Ready and RX-Empty signals from UART-Interface to the Arbiters.

Command-Decoder Decodes received instruction into command and address. Performs some interpretation to decide which Arbiter to notify.

Write-Arbiter Performs writing tasks and contains a de-serializer writing bytes received by UART-Interface to addressed register. While de-serialization is in progress, the process may be cancelled without side-effects by a reset or write command to a different address. Register access is done via ready-valid signaling and communicated via the Debug-Interconnect. As a result, the Write-Arbiter is able to wait until write-transaction has succeeded or transaction is cancelled by host.

Debug-Interconnect Multiplexing component allowing write access to connected registers. Implements a variant of the AMBA4-Lite protocol. Write data and Valid-Signal are multiplexed to the currently addressed register with the Ready-Signal routed back to the Write-Arbiter. Addressed components may indicate unavailability to write or read by keeping Ready-Signal or Valid-Signal lowered. Checks each cycle, if any connected register indicates new data through a set valid-signals. Address of such a register is sent to the Read-Arbiter.

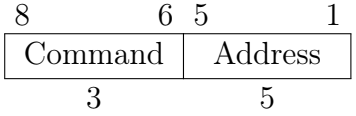
Read-Arbiter Performs all reading of registers over the Read-Interconnect via ready-valid signals. Unlike the Write-Arbiter, reading of registers cannot be cancelled once serialization into bytes has begun. Can continuously send content of addressed register to host, so long as the register signals validity of read data. If

idle, may send a command containing address of a register signaling availability to the host. This address is provided by the Read-Interconnect. If source address changes, a instruction containing the new address is sent to the host.

Instruction Format

As described in the chapter 5, commands are issued to the UART-TAP through prepending an escape character. Choice of character value must take potential conflicts with instructions into account, as repeating the escape character will instead cause the sequence to be interpreted as a single data word with value of the escape character.

Figure 6.6: Instruction Encoding



Code	Command	Description
0b000	CMD_NOP	Executes no operation and is used TAP internal to denote idling.
0b001	CMD_READ	Issues full read of the register addressed in received command, sending contents to host.
0b010	CMD_CREAD	Issues continuous read of the register addressed in received value. TAP will read and send register register content to host. Sending is paused if register signals unavailability and is canceled upon receiving another read command.
0b011	CMD_WRITE	Activates writing incoming data words to register addressed by command. Running de-serialization may

Continued on next page

Continued from previous page

Code	Command	Description
		be canceled by issuing write command with zero address.
0b100	Unused	
0b101	Reserved	Reserved to prevent potential repeat of escape character.
0b100 - 0b110	Unused	
0b111	CMD_RESET	Halts and cancels all read and write operations.

Address	Register (Name)	Description	Length (Bit)
0x00	NOP	Reserved. No register is mapped to this address. Used internally as default address	0
0x01	IDCODE	**IDCODE** register, equivalent to Debug-STandard.	32
0x10	DTMCS	**DTMCS** register, equivalent to Debug-Standard. Writing will have no effect at this point in time.	32
0x11	DMI	**DMI** register. equivalent to Debug-Standard and RISCV-Debug.	W:41 R:34
0x14 0x16	STB0_CS STB1_CS	Control & Status register for Streaming-Trace-Buffer. See chapter 6.2 for details	8
0x15 0x17	STB0_D STB1_D	Data register for Streaming-Trace-Buffer. See chapter 6.2 for details	32

6.1.3 DMI-Interface

For similar reasons as before, the DMI-Interface component had to be rewritten in order to interface with the TAP. Previously, relevant JTAG-signals had been multiplexed into the module by the original DTM if addressed, with the TAP being part of the DMI-Module. As the de-serialization is now handled by the TAP, the new DMI module has been made a peripheral to the TAP interconnect. The bi-directional Ready-Valid interface between the components allows the DMI to enforce waiting of the TAP until the transaction with the DM has finished.

6.2 Streaming Trace Buffer

While communication paths between core and FPGA are usable for the purpose of debugging, dedicated infrastructure is necessary to keep our goal of functional independence. With the existing data path through the TAP, an appropriate interface had to be developed, bridging the gap between FPGA and system clock domains. Hypothetically, design of such an interface could be deferred to the user by exposing the necessary signals of the TAP to the FPGA. This is advantageous in scenarios where the user's design requires no further debugging, rendering such architecture redundant. Yet, when debugging is required, this solution has drawbacks:

- Implementing infrastructure for debugging in soft-logic requires resources in addition to those of the actual design, limiting design size. Also, hard-logic implementations are more resource efficient.
- Errors in the user's debugging infrastructure are cannot be debugged by the infrastructure itself.

Opposed to this, an implementation in hard-logic can simplify user-designs and reduce overall cost. Requirement to said resource efficiency, however, is an implementation generalized and configurable enough to provide utility in a range of use cases. To this end, the Streaming Trace Buffer (STB), a peripheral to the TAP, has

been developed. The module, among other features, allows for capturing traces in dedicated SRAM, or using the same memory to facilitate streaming of data between the Host and the FPGA fabric.

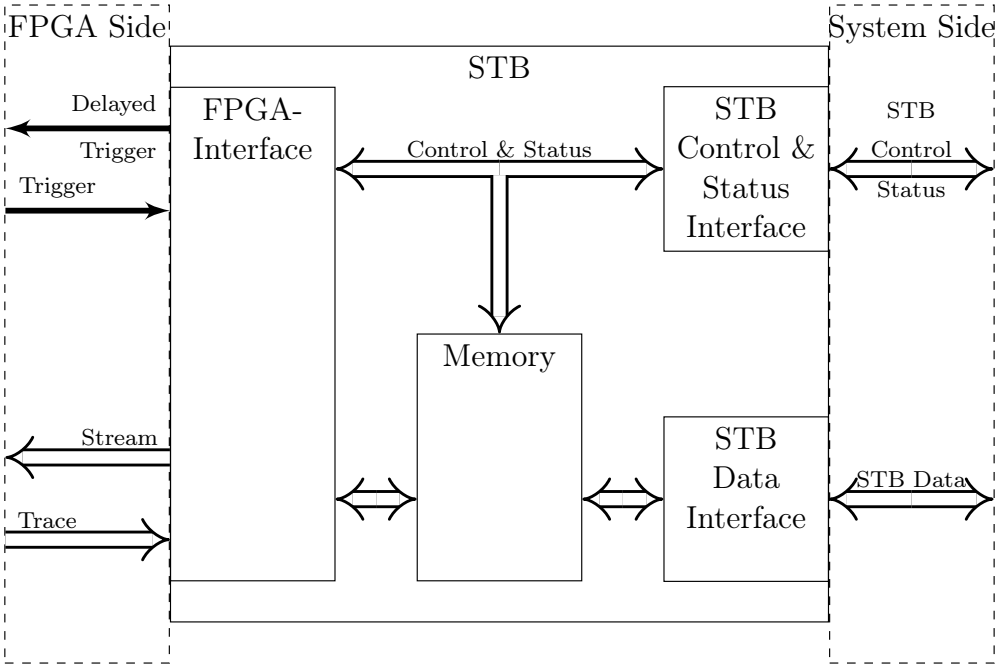


Figure 6.7: Data flow and logical division of the Streaming Trace Buffer.

Figure 6.7 shows the logical components of the STB module as well as data flow. From the left, the FPGA interface holds the trigger logic, handles trace capture and data transfer with the FPGA. This data is then stored in internal memory of the STB. From the other side, the system interface has access to the status and control registers and allows data exchange with the STBs memory.

6.2.1 System-Interface

The system interface consists of two RV-Registers, which contain state-machines to handle data exchange via the ready-valid interface of the TAP. As read and write data can come from different sources, aggregate signals like status and control can be accessed by the same instance of the RV-Register, saving resources and TAP-addresses.

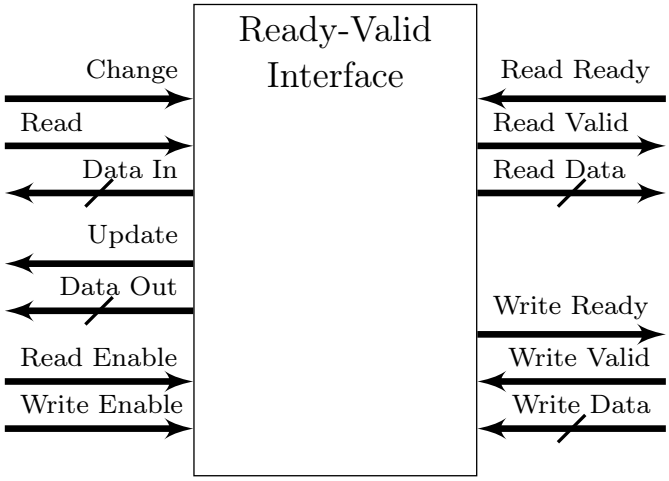


Figure 6.8: Ready-Valid Register Module

The interface offered to the device consists of a read & write enable, which allows the device to suppress reading and writing by the system. This is desirable in case of pointer collisions between the FPGA- and System-Interface, preventing data loss and signaling the TAP presence of back-pressure. The device, on the other hand, is able to assert a read-valid signal through a pulse on the change-port, independent of the intend by the TAP. This may signal the TAP the occurrence of an trigger-event and or new data to read. Successful reads and writes to the interface are signaled to the device via pulse of the update and read signals.

STB Control and Status

In order to reduce the number of required addresses by the TAP, both control and status registers are accessed by the same address. Writing to this address changes bits associated with control while reading will return status bits.

Figure 6.9: Control Register

8 5	4 3	2 1
delay	numtraces	mode
4	2	2

Table 6.3: Control Register Fields

Field	Description	Access	Reset
mode	Controls STB mode of operation: - 00b : Trace-Mode - 01b : RW Stream-Mode - 10b : Read-only Stream-Mode - 11b : Write-only Stream-Mode	W	0
numtraces	Number of traces captured in an FPGA cycle. Actual number is 2 to the power of numtraces.	W	0
delay	Controls percentage of memory allocated to traces before and after trigger by delaying write protection of memory by Trace-Logger. A high delay value skews ratio towards traces post-trigger. Unsigned integer.	W	15

Figure 6.10: Status Register

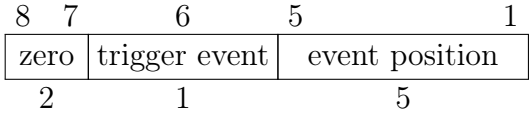


Table 6.4: Status Register Fields

Field	Description	Access	Reset
trigger event	Indicates a trigger event and associated elapsed delay. Only relevant during Trace-Mode and. Must be high to allow TAP to read data.	R	0
event position	Contains bit position on which trigger first occurred.	R	0

STB Data

Trace or stream data can be exchanged with the STB via the data register. However, during trace-mode, the data register can only be read after the delayed trigger event. Whether the register can be read or written in Stream-Mode depends on position of the read and write pointers of the FPGA-Interface and System-Interface respectively. Each read and write to the register will automatically advance respective pointers. A full readout of the SRAM memory requires therefore a number of reads equal to the SRAM depth.

Figure 6.11: Data Register

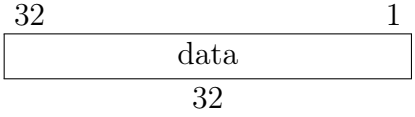


Table 6.5: Data Register Fields

Field	Description	Access	Reset
data	Register used for data exchange with the TAP. Is read only while STB is in read-only Stream-Mode or Trace-Mode.	R/W	0

6.2.2 FPGA-Interface

The FPGA-Interface module performs several important tasks:

- Serialization of traces words writeable to memory.
- Streaming of words to FPGA at variable rate
- Handling the pre-trigger condition and freezing capture of new trace data.
- Storing data in and loading data from the Memory-Controller.

- Keeping appropriate read and write pointers to memory.

These tasks may be categorized into functions of Tracing, concerned with (de-)serialization and trigger logic, and functions of Logging, concerned with memory transactions.

CDC Design

As FPGA and system reside in different clock domains, deciding which part of the logic is placed where is crucial to ensure correctness and effective use of resources. The following functional divisions have been considered during development.

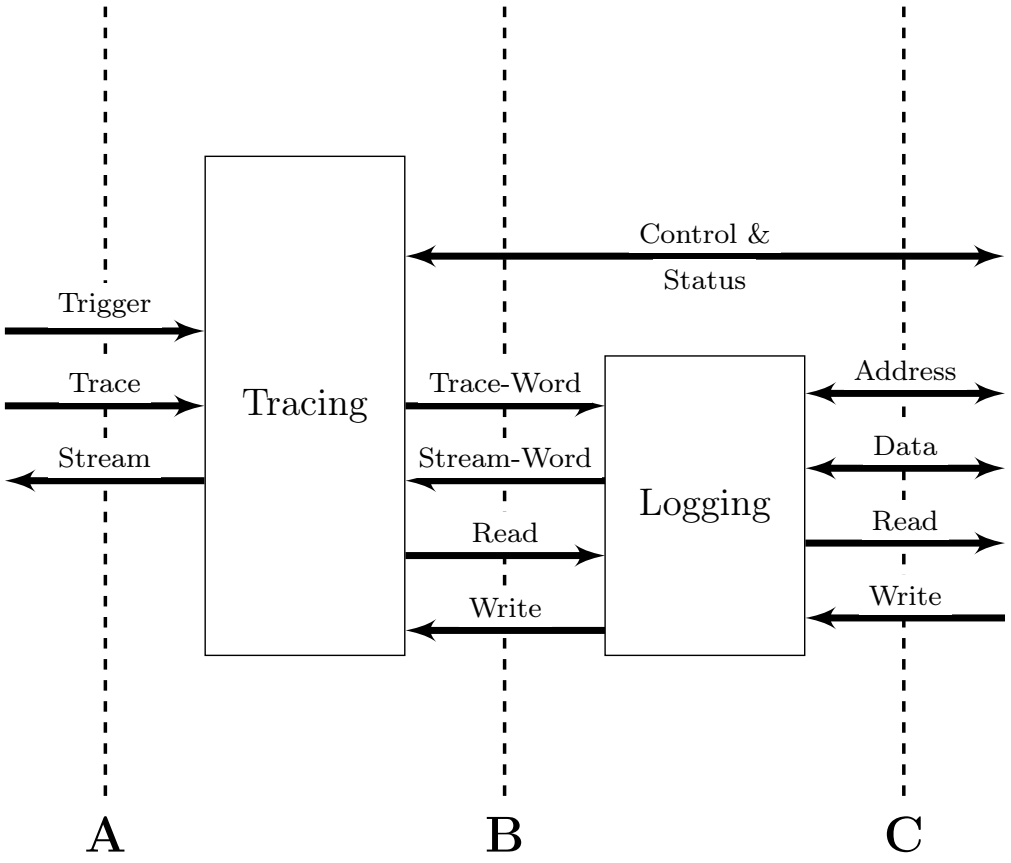


Figure 6.12: Different clock domain partitions of the STB.

A. **CDC between FPGA and -Interface**: The FPGA-Interface and all accompanying logic is placed in the system domain. Trigger signals, trace and stream data have to be synchronized from and to the FPGA.

B. **CDC between Tracing and Logging**: (De-)serialization and trigger logic is placed within the FPGA clock domain, memory transactions are handled in the system domain.

C. **CDC between Memory and FPGA-Interface**: FPGA-Interface is placed fully in FPGA-Domain. Control and Status signals with Memory Transaction have to be synchronized between domains.

As synchronization techniques all require a signal to be stable for a period of at least 1.5 times of the target domain, the first option is not viable in our case. With the FPGA clock being as fast as the system, OL-Synchronizers are unable to ensure accuracy of captured data. While it remains a possibility implement logic to stop the FPGA until the trace is acquired, such a function represents a large interference with the workings of the implemented design and goes counter to the goal of observability.

The third option does not suffer from such drawbacks. Serialization of the trace into words, which are writable to memory, is placed in the FPGA clock domain. So long as the rate of serialization does not exceed the time required by the synchronization, no trace-data will be lost even if FPGA and system clock domain are equally fast. However, as the Memory-Controller resides in the system clock domain, read and write pointers by the FPGA-Interface need to be synchronized between them.

The second option has the advantages of the third, but reduces the signals crossing the domain, making it the chosen solution.

The remaining signals, control, status, write and read data as well as respective signals indicating permission, can be categorized in the following groups:

Eventually accurate Permission signals, Load Request and Delayed Trigger fall under this category. All have in common, that their sampling does not need

to be cycle accurate as long as synchronization finishes before the next store or load event.

Sample accurate Control, Status, Trace- and Stream data consist all out of multiple bits which must be sampled at the same time.

Signals which are only required to be eventually accurate are synchronized through OL-Synchronizers. Sample accurate signals can be either synchronized using the MCP-Formulation or asynchronous FIFOs. However, as asynchronous FIFOs have a larger logic overhead, MCP-Formulation is chosen instead.

As changes in the Control signals are only valid after a reset, the reset signal has been chosen as the synchronization signal for the MCP-Formulation. For the Logger, which works on a granularity of memory words, changes in status need only be considered each time a new trace-word is stored. Hence, status and trace-data are synchronized together using one MCP-Formulation. Lastly, the Stream-data is synchronized using the grant signal.

Implementation

The Tracer module operates fully within the FPGA clock domain. It de-serializes trace data into memory words for the logger to process in one direction. In the other, stream data from the logger is serialized into the stream output. Exchange of data may happen at variable rates controlled by the "numtrace" register.

The Logger on the other end operates in the system clock domain. It handles data transfers with the Tracer over CDC and communication with the Memory Controller. This module handles its own set of read and write pointers to locations in the SRAM memory and contains logic to produce the delayed trigger event.

The trigger logic contained in the Trace-Logger module is kept simple to reduce logic cost. While it would be possible to design logic, which performs analysis on the trace data while also remaining configurable by the system, this task is better placed in the flexible FPGA fabric and hence deferred to the user's design.

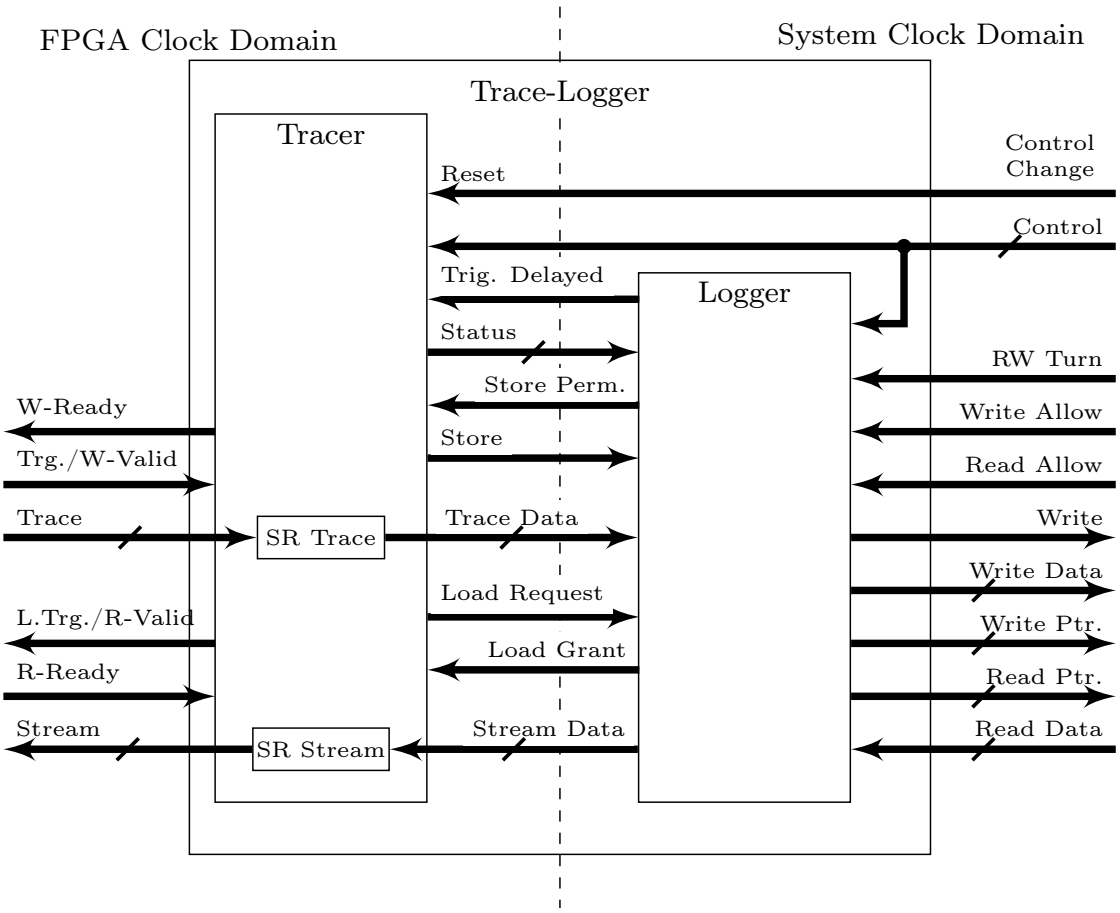


Figure 6.13: Trace-Logger Module

Table 6.6: Ports and Signals exposed to FPGA.

Port	Description	Reset
Trigger (Input, 1b)	Trigger port, which will cause writing trace data to memory to freeze after delay controlled by "delay". Is sticky, i.e. will start the timer on the first cycle the signal is set even if unstable. Will cause de-serialization position to be saved if asserted.	X
Trace (Output, 16b)	Trace input. Register "numtraces" controls rate of de-serialization, i.e. how many bits are of the input are de-serialized in parallel. Inputs with a index higher than the rate are ignored.	X
Write-Valid (Output, 1b)	Signals the FPGA side that writing to the trace is allowed. Only relevant in Stream-Mode to indicate whether data from the FPGA will be committed to memory	0
Read (Input, 1b)	Signals the Tracer a successful read of the stream and causes serialization to progress. Only relevant during Stream-Mode, ignored in Trace-Mode.	X
Stream (Output, 16b)	Stream output. Serialization rate is controlled by the "numtraces" register. In Trace-Mode, the data output is the memory word which will be replaced by the next write to memory and hence the oldest trace data. In Stream-Mode, data serialized originates from system.	0
Log Trigger (Output, 1b)	In Trace-Mode, this signal is set to high after the first occurrence of a trigger and a delay by the logger. Indicates freezing of writes to memory and can be used to daisy-chain multiple STBs for increased history depth. In Stream-Mode, this signal instead indicates validity	0

Continued on next page

Continued from previous page

Port	Description	Reset
	of the data output by the Stream-Port.	

6.2.3 Memory

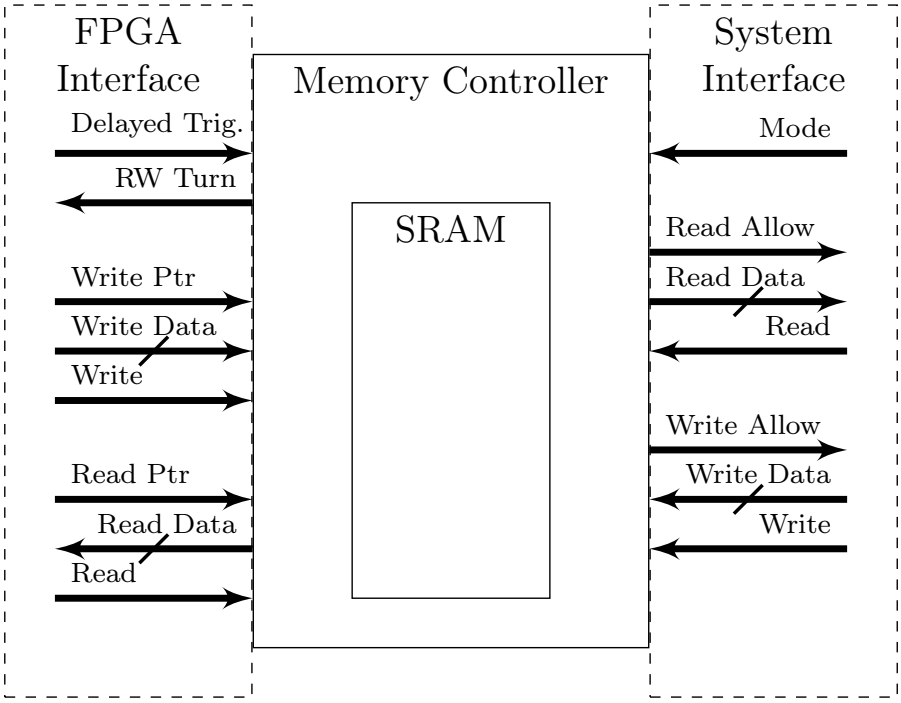


Figure 6.14: Memory Controller with internal SRAM

The memory cell primitive used by the FABulous SoC project has one read and write port, supporting independent operation. However, both system- and FPGA-interface need to perform independent read & write operations. To this end, the Memory-Controller implemented issues alternating RW-turns to the 2 participants. Further, the Memory-Controller handles the system read and write pointers and compares them against those of the FPGA-Interface. Depending on the mode of operation, the controller is able to rescind read and or write permissions for both interfaces to prevent data loss.

6.2.4 Operation

The STB features two distinct modes of operation, changing the meaning and purpose of signals on the FPGA-Side. Changing modes by writing to the control register always causes both the FPGA-Interface as well as the Memory-Controller to reset, rendering all data stored in memory effectively invalid.

Trace-Mode

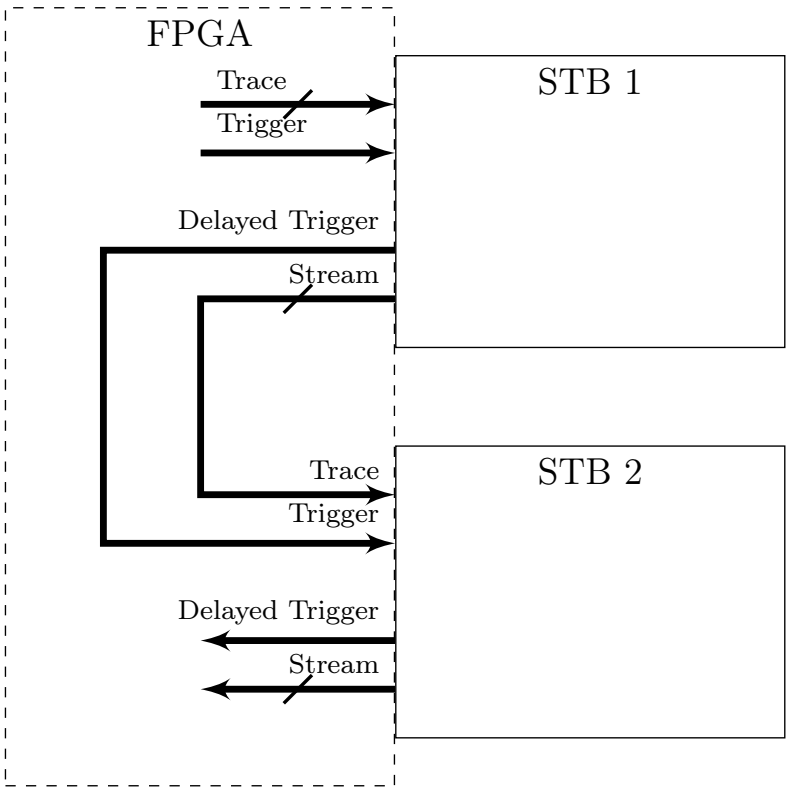


Figure 6.15: STB Daisy-Chaining

In Trace-Mode, the FPGA-Interface writes continuously to the Memory. Displaced words are output via **stream**, which allows daisy-chaining of two STBs. Fig.6.15 visualizes this. As data output by **stream** is oldest data contained in the memory, routing it to the trace input of the second STB allows using its memory to extend the trace history. Similarly, the delayed trigger is connected to the trigger port of the second STB, causing it to halt memory store operations if configured

correctly. To make use of this feature, the rate at which memory words are loaded may not exceed the rate at which load requests can be served. As requests from the Tracer sub-module must travel over CDC to the Logger, it can require between 4 to 5 cycles in the system clock domain until the requested data arrives. Assuming both system and FPGA run at the same frequency, no more than 4 parallel traces may be captured at a time. Slower FPGA clocks allow for higher numbers parallel numbers while maintaining ability for daisy-chaining.

Upon successful trace acquisition and assertion of the trigger condition, the trace history can be read out from the data-register. The position of the word in the read-out, on which the trigger happened, can be determined by the "delay" field of the control and the "bit position" field of the status register. The formula to determine the trigger position A in the readout is as follows:

$$\mathrm{A} := (D-1)(1-\lfloor\frac{n}{N}\rfloor) \tag{6.3}$$

With n being "delay" ranging from 0 to 15, D the memory depth of 32 and N being maximum value.

Table 6.7: Word position A in read-out over delay n.

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	31	29	27	25	23	21	19	17	15	13	11	9	7	5	3	0

This, with the number of traces and the bit position from the status register, allow full reconstruction of the trace from the readout.

Stream-Mode

To facilitate data exchange between system and FPGA, the STB features three Stream-Modes: bi-directional read & write, system read-only, and system write-only. All modes make use of the STBs internal memory to buffer communication. Central to this function is monitoring of pointer collisions between system- and FPGA-side.

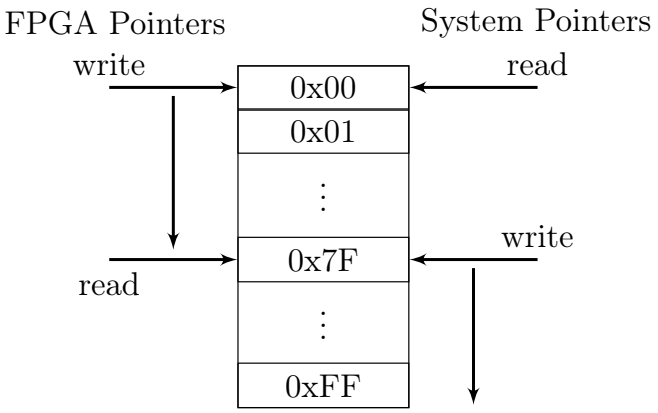


Figure 6.16: Read & Write pointer envelopes on internal STB-Memory addresses.

In read & write mode, the read and write pointers of the FPGA-Interface as well as the respective pointers by the system interface must fulfill the following conditions.

1. Read pointers may not overtake write pointers of the opposite interface.

2. Write pointers may not overtake read pointers of the same interface.

If a read or write from any interface would violate these conditions, the Memory-Controller will rescind appropriate permission of the offending interface. While the bi-directional read & write mode should cover a wide range of applications, in the event of a large mismatch between incoming and outgoing data rates, the STB may also be dedicated to one direction. This allows a larger part of the internal memory to be used for buffering data from one source and achieve larger burst sizes. In the read only Stream-Mode, FPGA-Interface only possesses write privileges while the System-Interface read privileges, vice versa in write-only Stream-Mode. In either mode, the memory controller will ignore unused read and write pointers from both interfaces.

7 Software Implementation

Central to the debug support on the host system is the OpenOCD project[**rath**], which bridges the gap between the device and debugger software by providing an abstract interface for the latter. Additionally, the project supports commands for loading data into memory locations specified by parameters, and, to a limited extend, support configuration of FPGAs natively. However, as capabilities change from device to device, OpenOCD relies on configuration files to denote supported functions of the target device. Several interfaces are supported to carry the debug information and commands, however, of these supported debug-transports, UART is not included. The integration of the proposed UART-debug-communication protocol into OpenOCD has been considered, though, given the scope of the task as well as compatibility concerns with future versions, another solution was preferable: A translation Adapter between JTAG and the UART-Debug-Protocol.

7.1 JTAG-to-UART Adapter

Adding a layer of abstraction between OpenOCD and the physical device the JTAG-to-UART Adapter is a small piece of software handling the specific communication with the UART-DTM and its protocol while providing a virtual JTAG-Interface for OpenOCD. Basis for the Adapter program is the feature of OpenOCD to package JTAG signals into a remote bitbang protocol communicating over TCP/IP. This has the side effect of allowing potentially debugging of a device connected remotely. The communication process can be seen in Figure 7.1.

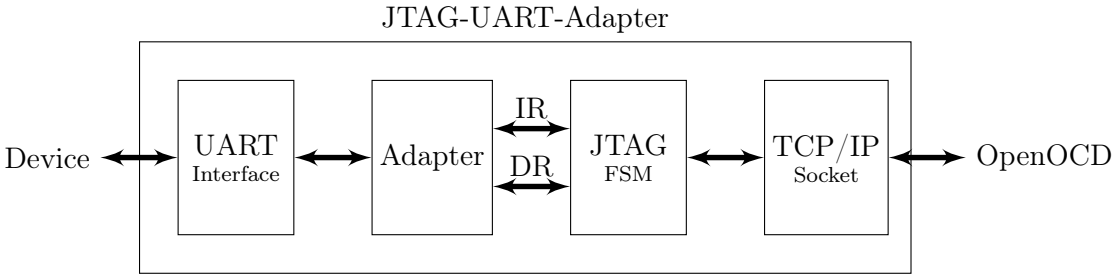


Figure 7.1: Translation process of the JTAG-to-UART Adapter.

Opening up a TCP/IP socket, the Adapter interprets the bitbang protocol an feeds the supplied signals **TMS**, **TCK** and **TDI** into an internal implementation of the JTAG state-machine. Using the address provided by the virtual JTAG-registers **IR** and data of the **DR** register, write and read commands to the addressed DTM-Registers with (de-) serialization of the data in and output is performed. As JTAG is synchronous, the actual Adapter class has to perform a read at the earliest possible opportunity to provide actual register contend without delays visible to OpenOCD. Similarly, full shifts through the data register **DR** triggers an exchange of data between the Adapter and the Device. As such, progress in the JTAG-State-Machine is limited by the latency of sending data to the DTM followed by requesting a read and receiving the answer. Given this process, the Adapter is not capable of making use of full extend of the streaming capabilities.

7.2 Simulation Tools

In order to simulate the CVA6 core with the implemented debug architecture the open source simulation program Verilator[**zotero-254**] was used. As the CVA6 project already implements a testbench for Verilator, only the implemented modules had to be added. With the original implementation of debug infrastructure using JTAG, the projects supplies a Direct Programming Interface (DPI) which provides means for software external to communicate with the simulated model. While a DPI exists for JTAG making use of the bitbang TCP/IP interface of OpenOCD, no DPI

existed for UART. For the simulation of the UART-DTM, a DPI was implemented providing a virtual serial interface to external programs. Internal to the simulation, frames are serialized into **RX** and de-serialized from **TX** at a specified baud rate. This allows for the testing of the UART-Interface implementation.

8 Implementation Analysis

The proposed architecture was tested on a Nexys-4 FPGA with a system-clock of 25 MHz and a baud rate of 3M baud. Setup of communication with OpenOCD was successful. Although unreliable, OpenOCD was able to start the RISC-V and load a test program into system memory. The exact reasons for this problem could not be determined at the time of writing, but, as OpenOCD communicates in synchronous fashion, the low reliability may stem from the low data rate through communication latencies. Isolated tests with the DTM implemented on the FPGA revealed full functionality of expected functions, placing the likeliest source of the issue with the Adapter. Streaming data to the STB via continuous writing yielded the expected raw data rate achievable through the UART interface with set baud rate. Stalling due to back pressure did not occur but could be provoked.

Table 8.1: Area requirements by STB and components in Skywater 130nm process with LUTs and registers of implementation with Artix-7 as target.

Component	Area (μm^2)	LUTs	Registers
With CDC techniques:			
STB	184204.5	297	390
- Tracelogger	12406.6	248	306
- Memory-Controller	171632.5	45	76
– SRAM-Cell	167999.0	20	50
- RV-Interfaces	217.8	4	6
Without:			
STB	182511.7	175	304
- Tracelogger	10682.7	245	220

Table 8.1 shows the area required by the STB and constituent components. LUTs and registers of an equivalent implementation for an Artix-7 FPGA are provided for

comparison. The values were acquired by running the open-source synthesis framework Yosys[**wolf**] using the Skywater 130mm Digital Standard Cells, and SRAM Macros. By far the greatest area is required by the SRAM-Cell, making the surrounding logic small in comparison. Area used by the CDC crossing techniques contained in the Tracelogger module represent an increase of ≈20% compared to without. This increase is predominantly caused by the additional registers, as the resource costs of the FPGA implementation highlights.

Table 8.2: LUT and register cost of Debug-Transport from implementation on with Artix-7 FPGA.

Component	LUTs	Registers
Proposed DTM		
UART-DTM	331	401
- UART-Interface	95	123
– RX-Module	29	45
– RX-FIFO	26	38
– TX-Module	40	39
- UART-TAP	192	120
- Escape-Filter/Filler	14	22
- Debug-Interconnect	21	50
- UART-DMI	9	86
Original DTM		
JTAG-DTM	175	341
- DMI-CDC	46	164
- JTAG-TAP	128	93

Table 8.2 shows the required FPGA resources of the original JTAG-DTM. Due limited support for SystemVerilog by Yosys, area numbers couldn't be acquired at the time of writing. However, the number of LUTs and registers can still be used as a measure of complexity. In this regard, the UART-DTM is almost twice as expensive regarding the number of LUTs, while increasing required registers by ≈17%. Although the UART-TAP itself is about as expensive as the JTAG-DTM with its costliest component being the CDC, the costs are nearly doubled through the UART-Interface alone. Comparing the total required LUTs and registers with

the corresponding area size from the STB, required area of the UART-DTM (without the SRAM-Cell) will likely be of the same order due to similar complexity.

Table 8.3: LUT and Register costs of Debug-Module from implementation on Artix-7 FPGA.

Component	LUTs	Registers
Debug-Module	455	676
- DM-CSRS	278	599
- DM-MEM	100	74
- DM-SBA	77	3

Using the values from table 8.3, 8.2 with the area costs from table 8.1 as basis, the required area for the entire debug infrastructure can be estimated. With the STB without the SRAM as baseline, the DM is about twice as complex as the STB. Assuming similar implementation sizes of the required cells, the implemented DM with UART-DTM would likely require an area around 3 times the baseline. Considering the size of the STBs logic compared to the SRAM, the size of the SRAM can be used as an upper limit. With a chip size of 10 mm^2, DM with UART-DTM would take up the minimum of 0.5% and a maximum of 5%. Adding two STBs with SRAM will take up approximately at worst 8.7%, leaving more than 90% of the area for other components.

9 Future Work and Conclusion

The communication path between OpenOCD and the DM does not make full use of the features provided by the debug communication protocol and supporting hardware. Further work includes a better software support potentially re-implementing features previously supplied by OpenOCD in the Adapter software or integrating the protocol into OpenOCD itself. From the other end, deeper integration of capabilities provided by the protocol into the DM could also allow asynchronous messaging by the module and could help facilitate further features involving monitoring of the system bus as well as hardware triggers.

However, with the goals formulated in the beginning, the implemented debug architecture supports all desired features or is extendable enough to support them in the future. From a users point of view, a Host is able to query and control State of CPU, memory can be read an written while Trace-capture and data exchange with the eFPGA is possible. All those features remain usable independent of the CPU and FPGA, allowing dedicated use of either resource. From the point of view of a hardware designer, the proposed architecture requires only two package pins while remaining small and modular to allow implementation of additional features. As all tools used are open-source, both the tools and the implementation fulfill the goals of traceability and support educational uses.

Part I

Appendix

A Lists

A.1 List of Figures

A.2 List of Tables

B Bibliography

[1] Clifford Cummings. "Clock Domain Crossing (CDC) Design & Verification Techniques Using SystemVerilog". In: (Jan. 2023).

[2] *CVA6 RISC-V CPU*. OpenHW Group. Aug. 2022. URL: https://github.com/openhwgroup/cva6 (visited on 08/03/2022).

[3] *Devicetree Specification Unknown-Rev — Devicetree Specification Unknown-Rev Documentation*. URL: https://devicetree-specification.readthedocs.io/en/latest/index.html (visited on 01/14/2023).

[4] *GitHub - Nguyendao-Uom/eFPGA_v3_caravel: https://caravel-user-project.readthedocs.io*. URL: https://github.com/nguyendao-uom/eFPGA_v3_caravel (visited on 03/15/2023).

[5] Dirk Koch et al. "FABulous: An Embedded FPGA Framework". In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '21. New York, NY, USA: Association for Computing Machinery, Feb. 2021, pp. 45–56. ISBN: 978-1-4503-8218-2. DOI: 10.1145/3431920.3439302. URL: https://doi.org/10.1145/3431920.3439302 (visited on 01/13/2023).

[6] *Nexys4DDR User Guide*.

[7] *Overview of Skywater Technology 130nm Process*. URL: https://stineje.github.io/pages/overview.html (visited on 01/11/2023).

[8] *Qemu/Target-Riscv.Rst at Master · Qemu/Qemu*. URL: https://github.com/qemu/qemu (visited on 07/16/2022).

[9] *RISC-V Debug Support for PULP Cores*. pulp-platform. July 2022. URL: https://github.com/pulp-platform/riscv-dbg (visited on 07/19/2022).

[10] *SpinalHDL/VexRiscv*. SpinalHDL. July 2022. URL: https://github.com/SpinalHDL/VexRiscv (visited on 07/18/2022).

[11] *Virtex-II Pro Data Sheet*. June 2011. URL: https://media.digikey.com/pdf/Data%20Sheets/Xilinx%20PDFs/Virtex-II_Pro_Pro_X.pdf.

[12] Andrew Waterman, Krste Asanovic, and CS Division. "Volume I: Unprivileged ISA". In: (), p. 238.

[13] Andrew Waterman et al. "Volume II: Privileged Architecture". In: (), p. 155.

[14] F. Zaruba and L. Benini. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-Nm FDSOI Technology". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (Nov. 2019), pp. 2629–2640. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2019.2926114.

[15] *Zedboard Hardware Users's Guide*. URL: https://www.avnet.com/wps/wcm/connect/onesite/922900e3-3d57-4cc7-883f-a8b9fbea0cd0/ZedBoard_HW_UG_v2_2.pdf?MOD=AJPERES&CACHEID=ROOTWORKSPACE.Z18_NA5A1I41L0ICD0ABNDMDDG0922900e3-3d57-4cc7-883f-a8b9fbea0cd0-nxyWMFS (visited on 01/01/2023).

[16] "Zynq-7000 SoC Technical Reference Manual". In: (2021).

Erklärung:

Ich versichere, dass ich diese Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den (Datum) .....