Karlsruhe Institute of Technology

Institut für Technische Informatik
Chair for Embedded Systems

Implementation of a Memory Access Trace Unit for a RISC-V SoC

Bachelorarbeit
von

Manuel Killinger

am Karlsruher Institut für Technologie (KIT)
Fakultät für Informatik
Institut für Technische Informatik (ITEC)
Chair for Embedded Systems (CES)

Erstgutachter:	Prof. Dr. Jörg Henkel
Zweitgutachter:	Prof. Dr. Wolfgang Karl
Betreuer:	Dipl.-Inf. Paul Genssler, Dr. Lars Bauer
Tag der Anmeldung:	12.05.2020
Tag der Abgabe:	11.09.2020

Erklärung

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Die verwendeten Quellen und Hilfsmittel sind im Literaturverzeichnis vollständig aufgeführt.

Karlsruhe, den 11.09.2020

Manuel Killinger

Abstract

One of the most important aspects of testing and analyzing embedded systems is the system monitoring itself. Newly developed System on a Chip can be verified and tested using simulation software, with the drawback of often being computationally expensive or slow. Especially of interest is the read and write behavior of SoCs. This work presents the implementation of a Trace Unit, developed for the Freedom U500 SoC. It is a non-invasive component that can monitor all memory accesses flowing to and from the main memory. The Trace Unit is integrated into U500's memory subsystem to expose the access behavior during the execution of various benchmarks, and the results are collected and presented.

Zusammenfassung

In dieser Arbeit wird die Implementierung einer Speicher Trace Einheit vorgestellt, die für das Freedom U500 System on a Chip (SoC) [SiF16] entwickelt wurde. Sie ermöglicht es Speicherzugriffe des im SoC integrierten RISC-V SiFive U5 Prozessors aufzuzeichnen, zu verarbeiten und an einen über PCIe angeschlossenen Host Computer zu senden. Die verschiedenen Komponenten innerhalb des SoC sind über das TileLink (TL) Protokoll unter Verwendung von Diplomacy [CTL17], einem System zur automatischen Parametrisierung des TL Netzwerks, miteinander verbunden. Die Trace Einheit unterstützt die Parameter Inferenz, so dass sie an beliebiger Stelle als passiver Knoten in das Netzwerk eingebracht werden kann um alle Transaktionen über jene Verbindung abzugreifen. Zur Analyse des Speicherzugriffsverhalten des SoC wurde die Trace Einheit zwischen den Speicher Bus und den Speicher Controller eingebracht. Unter Anwendung verschiedener Benchmarks wurden Daten erfasst, verarbeitet und die Ergebnisse präsentiert.

Contents

AMBA4 ARM Advanced Microcontroller Bus Architecture 4

AXI4 Advanced eXtensible Interface 4

CBUS Control Bus

CDC Clock Domain Crossing

Chisel Constructing Hardware in a Scala Embedded Language

CISC Complex Instruction Set Computer

CLINT Core Local Interruptor

CU Control Unit

DMA Direct Memory Access

EOF End of File

FIRRTL Flexible Internal Representation for RTL

FPGA Field Programmable Gate Array

HDL hardware description language

ISA Instruction Set Architecture

MBUS Memory Bus

PBUS Periphery Bus

PCIe Peripheral Component Interconnect Express

PLIC Platform Level Interrupt Controller

RISC Reduced Instruction Set Computer

Rocket Rocket Chip Generator

RV RISC-V

SBUS System Bus

SoC System on a Chip

Tcl Tool Command Language

TL-C TileLink Cached

TL TileLink

TL-UH TileLink Uncached Heavyweight

TL-UL TileLink Uncached Lightweight

TU Trace Unit

U500 Freedom U500

VC707 Virtex-7 FPGA VC707 Evaluation Kit

List of Tables

List of Figures

Chapter 1

Introduction

Semiconductor fabrication has become increasingly difficult as the physical properties of silicon are becoming limiting factors. This leaves leading companies struggling with their transition to smaller feature sizes to further increase the performance and energy efficiency of new processor generations while keeping them profitable. While the desktop and notebook market is still dominated by processors implementing the x86 Instruction Set Architecture (ISA), the struggle to increase transistor density combined with the rapid growth od the smartphone market gave other architectures the chance to catch up. The steady performance increase of smartphones has proven that implementations of ARM's instruction set can deliver great performance while being energy efficient. ARM processors have dominated the smartphone market for a couple of years now, and with Apple's announcement to ditch Intel's x86 processors for ARM designs in their upcoming notebooks and tablets, the reigning x86 architecture seems to be slowly pushed out of the handheld market. Complex Instruction Set Computer (CISC) designs were undoubtedly a perfect choice in the 1980s, because of their design philosophy. They can execute compact code at the expense of more extensive decoding logic. This was the driving advantage back then when memory was expensive, and the physical limits of semiconductor fabrication had not been reached yet. Now that memory has become significantly cheaper, older architectures suffer from their choice of the instruction set and backward compatibility, spending precious die area on rarely used logic, which hinders advances towards energy-efficient designs. Modern SoCs also tend to feature various co-processors for artificial intelligence or video processing, instead of relying on the CPU and GPU as sole sources for processing power. This development makes Reduced Instruction Set Computer (RISC) designs very appealing as a choice for a modern ISA and further research. A very attractive platform is the RISC-V ISA, developed by UC Berkeley. Thanks to the permissive BSD license, anyone can design, manufacture, and sell RISC-V microprocessors. Implementations of the ISA are already used in education, academia, and commercial applications. RISC-V (RV)'s support for custom instruction set extensions makes it perfect for the development of SoC with customized co-processors.

Memory tracing plays an essential role during the development of embedded systems. Currently, embedded systems are mostly simulated, and therefore tracing memory accesses via simulator is slow and requires a lot of processing power, especially in larger SoC designs. This work presents the Trace Unit (TU), a memory access tracing component that can be introduced into the Freedom U500 (U500) SoC [SiF16] to capture all memory accesses that are sent from the SoC's integrated SiFive U5 core to main memory. It allows for

data acquisition in real-time, sending the data via Peripheral Component Interconnect Express (PCIe) to a connected host for further analysis. Using the TU can give insight into the memory access behavior of the system running on the U5 processors, such as read and write patterns, throughput, and access latency. Chapter 2 will provide background on the technologies used for the development, chapter 3 explains the overall design, whereas chapter 4 provides specific implementation details. The results of custom benchmarks are presented in chapter 5.

Chapter 2

Background

This chapter will provide insight into the different technologies used in this thesis that are required to understand the following chapters. The following sections go into detail about the related hardware components used in the setup, providing a thorough explanation of the U500 [SiF16] and its processor, the RISC-V GC Rocket Chip Generator [AAB+16]. Conclusively, the TileLink protocol [SiF19] will be explained as is plays a significant role in the Rocket Chip and Trace Unit as well.

2.1 The Freedom U500 Platform

The platform of choice to develop the Trace Unit on is the SiFive Freedom U500 [SiF16]. It is the first addition to the SiFive Unleashed family, aiming to create a basis for highly customizable RISC-V SoCs. U500 supports up to eight 64-bit RISC-V GC cores, including customizable data and instruction caches, features a shared L2 cache that can be configured as a multi-bank layout or scratchpad and hooked up to DDR3/DDR4 DRAM, PCIe Gen 3.0 support, up to 1Gb Ethernet, USB3 3.0, a Platform Level Interrupt Controller (PLIC), an on-chip debug unit and support for several peripheral devices. Self-written components such as controllers for customized peripherals can be introduced into the design with little effort, and if necessary, the integration of specific co-processors that make use of the extensible RISC-V instruction set is supported as well. U500 is designed to be a fully customizable SoC, enabling it to be synthesized for several FPGA boards and is capable of running Linux. All components are tested by SiFive, and therefore U500 presents itself as an excellent choice for rapid RISC-V product prototyping, as well as academic use to develop and evaluate the performance of custom hardware like the Trace Unit. SiFive provides an implementation of the U500 SoC [Git] for the Virtex-7 FPGA VC707 Evaluation Kit (VC707) [Xilb], and from this point onward, further explanations and references to the U500 are always about this implementation. U500 consists of several components. Figure 2.1 depicts how they are connected to each other.

2.1.1 The U5 Core Complex

The primary source of processing power comes from four 64-bit U5 RISC-V Application cores. The base cores are generated using the Rocket Chip Generator [AAB+16]. It is a highly customizable SoC generator designed by UC Berkeley as an open source tool to create RTL of the RISC-V ISA and is now mostly maintained by SiFive. The generator

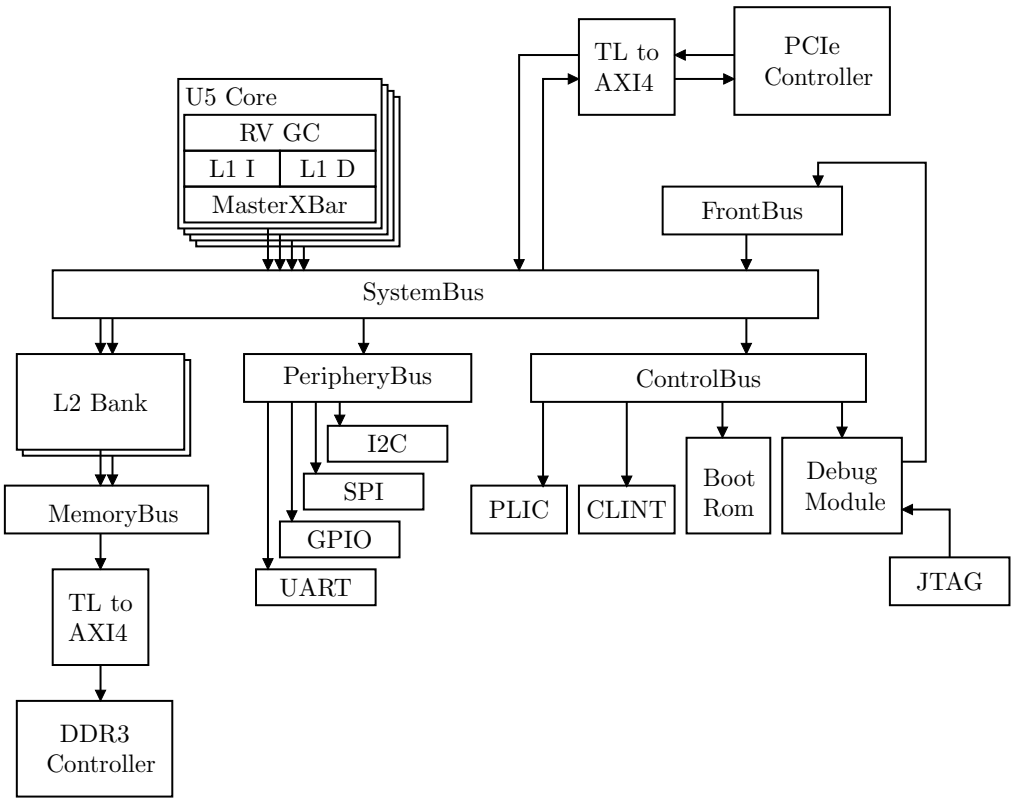

Figure 2.1: Freedom U500 implementation on the VC707 Evaluation Kit

can be configured to target cores that implement various combinations of the RV ISA extensions. All cores used in U500 support the standard Multiply, Single-Precision Floating Point, Double-Precision Floating Point, Atomic, and Compressed RISC-V extensions. They are also configured to have 16 KiB 4-way L1 I and D caches and 512 GiB virtual address space using the Sv39 virtual address translation scheme with 32 entry TLBs for both caches. SiFive provides those base cores with their JTAG Debug Module, Platform Level Interrupt Controller, and Core Local Interruptor to form the U5 Coreplex.

2.1.2 SiFive Shell

SiFive embedded the U5 cores into the design by attaching peripherals, created clock management and reset handling, and memory pipeline to connect the vc707s DDR3 memory to the memory bus. They introduced the PLIC and Core Local Interruptor (CLINT). Even though several Rocket subsystem components contain "bus" in their name, they differ from the traditional understanding of a bus. They are in fact implemented as TileLink crossbars. This means that connecting n masters to m slaves will potentially generate $n * m$ TL connections. Connecting the components in this way may generate very high fanout nets, but in return, it enables much higher throughput than a traditional bus where only a single device has access to the bus at a given time slot. The main bus that the cores are connected to is the System Bus (SBUS). The PCIe controller is also connected to it. The SBUS masters the Periphery Bus (PBUS). U500 has SiFive's I2C, SPI, UART and general purpose IO controllers connected to it. The Control Bus (CBUS) connects the Boot Rom, Debug Unit, and the PLIC and is also mastered by the SBUS. U500

supports a multi-bank L2 cache with multiple memory channels, but currently, this feature is disabled, resulting in the Memory Bus (MBUS) being directly connected to the SBUS. A PCIe controller is connected directly to the SBUS. Using th VC707 version with the expansion card [PCI] allows for multiple devices to be connected through this controller.

2.1.3 Board

The board hosting the U500 SoC is the Virtex-7 FPGA VC707 Evaluation Kit, and to provide Ethernet access a PCI Express Gen2/3 X8 Root FMC Module [PCI] is connected to it. A generic Ethernet networking card is inserted into the FNC module to allow for SSH connections to the Linux running on the RV.

2.2 TileLink Description

TileLink (TL) [SiF19] is an open-source bus protocol to interconnect multiple masters with multiple slaves through coherent memory-mapped address spaces, mainly designed for use inside the Rocket Chip, but not limited to that. It is divided into three different conformance levels, TileLink Uncached Lightweight (TL-UL), TileLink Uncached Heavyweight (TL-UH), and TileLink Cached (TL-C), where each former is a subset of functionalities of the latter, TL-C being the most powerful providing full support for cache coherence. Communication on a TL bus is done via transactions on up to five channels per connection between two nodes, namely a, b, c, d, and e. Channel a goes from master to slave to send requests. The d-channel goes from slave to master to return data, b from slave to master for probes, the c-channel from master to slave for releases, and the e-channel is used for grant acknowledgments. Going forward, only TL-UH will be elaborated on, as the connection, the TU will monitor only uses this protocol. The a- and d-channels are mandatory in all three conformance levels, whereas TL-C is also using the b- ,c- and e-channels. Messages put on the channels are called transactions, and TL-UH allows for bursts of varying sizes. The most important messages on the a-channel are PutFullData, PutPartialData, and Get. PutFullData is used to write data to a slave, possibly in bursts. PutPartialData allows for writes with a mast specified, and Get is used to request data. Put and Get requests are responded to on the d-channel using AccessAck and AccessAckData, respectively. The a-channel provides source and size fields. The source field is used to route responses back to the sender. The size field specifies the number of bytes to be written or read. If size is greater than the bus width, it indicates a burst read or write. Diplomacy [CTL17] is used to negotiate free parameters in the TL network. This way, bus widths are inferred for adapter nodes such as buffers or crossbars.

Chapter 3

Design

The Trace Unit (TU) is a non-invasive component that can be freely placed anywhere in the TileLink (TL) network to capture all transactions flowing between two connecting TL nodes. The captured data is processed and sent via PCIe to a connected host. On the host side, there are two device files exposed, namely */dev/xillybus_data* and */dev/xillybus_status*. The seekable status device file exposes the Trace Unit's status memory, allowing the modification of the status memory from the host. Writing any value greater than zero to address 0 enables the capture process, disabling it works by writing 0, respectively. When data capture is enabled, the memory trace can be read from the data device file. On disabling the capture, or when a pipeline overflow happens, an End of File (EOF) is generated on the data stream. This way, a reading application is notified that the data capture process has ended. Other data can also be extracted through the status memory, like if an overflow happened, or information about the TL channels monitored. This allows an analyzing application to process the data correctly. The data stream is in a format that is easily decodable for information extraction. More on the status memory is elaborated in section 3.5.

While the primary purpose of development lies in monitoring the U500 SoC's transactions to and from main memory, the TU is designed to be adaptable by inferring parameters from the TL Diplomacy network and is thus not limited to only monitoring the main memory channel, but can be introduced anywhere in the SoC. The focus on adaptability has implications on the choice of hardware description language (HDL) to describe the TU with, and on the integration of the TU into the SoC, as there are two options to choose from, developing it for the Advanced eXtensible Interface 4 (AXI4) or TL protocol. These topics are discussed in sections 4.1 and 3.1 respectively.

3.1 Hook into Memory System

At the beginning of the design process, there was a decision to be made regarding where is the best possible place to integrate the Trace Unit into Freedom U500's memory hierarchy. There are two protocols used in the SoC, namely the AXI4 and TileLink. The TU needs to act as a communication device that supports either one of those.

AXI4 is the reigning industry standard for on-chip communication and part of the ARM Advanced Microcontroller Bus Architecture 4 (AMBA4) [AMB] specification. It enables multiple masters to interface with multiple slaves over parallel high-performance, synchronous buses by offering support for unaligned data accesses, burst transfers, separate

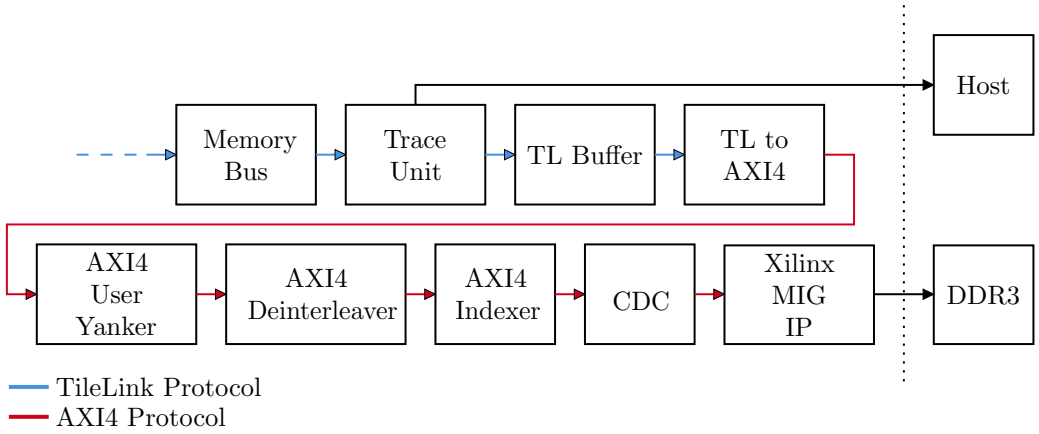

Figure 3.1: Freedom U500 Memory subsystem with placement of the Trace Unit

read and write channels, atomic instructions, and multiple outstanding transactions by monitoring thread IDs. TileLink [SiF19] on the other hand, is a relatively new protocol designed for use inside RV SoCs with the main focus on providing masters cache-coherent, memory-mapped access to multiple slave devices using a MOESI-equivalent protocol. It is used extensively inside U500 to connect all main components, and when necessary, TL is converted to AXI4 before attaching peripherals like the controllers for external memory. While the choice of either protocol allows for tracing of all memory transactions, TL is used way deeper inside the SoC and developing for TL enables monitoring of other connections, for example, by placing the TU between a single U5 core's connection to the SBUS. Therefore the decision was made to use TL. The choice of TL over AXI4 does not limit the TU's ability to track memory accesses on AXI4 connections, because the Rocket Chip library provides multiple adapters for protocol conversion from TL to AXI4 and vice versa. But monitoring AXI4 connections would possibly affect throughput due to the use of such adapters in the data path.

Figure 3.1 shows the memory subsystem starting from the MBUS up to the controller accessing the 1 GiB DDR3 off-chip. The is introduced directly after the MBUS and before TL transactions are converted to TileLink. Protocol conversion is performed using several modules that take care of the specific differences, and after crossing clock domains, the Xilinx Memory Interface Generator IP takes care of accessing the DDR3 RAM.

3.2 General Design

The Trace Unit itself acts as a TL node and wraps three main components. The Pipeline takes care of processing data coming from the TL connection monitored, the Control Unit (CU) enables and disables the design and keeps track of performance stats, and the PCIe component provides the interface to the Linux host. Figure 3.2 shows the general design. The main purposes of each of those components are elaborated in the following subsections.

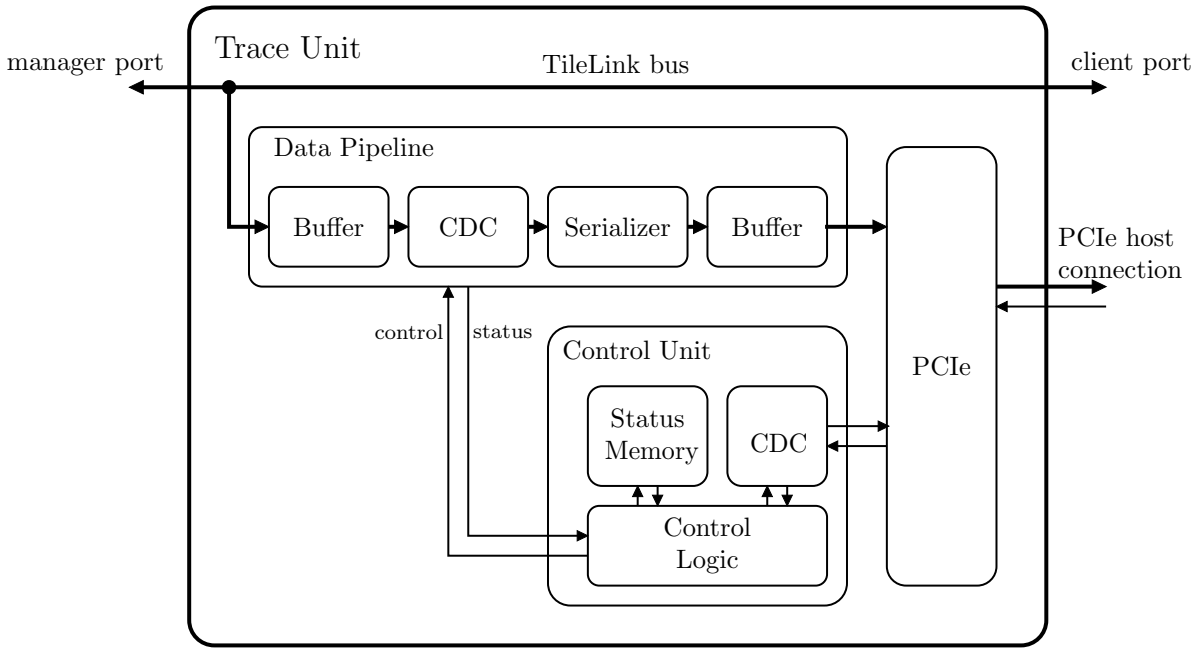

Figure 3.2: General Design of the Trace Unit

3.2.1 Data Pipeline

The Pipeline's primary purpose is to process the transactions that appear on the bus by splitting them into manageable sized chunks and sending them to the host through a PCIe controller. The Pipeline is arranged using multiple modules in succession connected using a valid ready interface, each serving a single purpose. Whenever any of the TL channels has a valid transaction accepted by the receiver, i.e. fires, a cycle count value is appended in front, and then the transaction is fed into the pre-crossing buffer. The buffer is necessary to balance out spikes in throughput on the TL bus, that would exceed the maximum throughput of the Serializer. That might be the case when the TU is introduced to monitor a connection that implements the TL-C conformance level and uses all five TL channels, or a parametrization of the bus is chosen that results in high bus widths of the TL channels. After the buffer a Clock Domain Crossing (CDC) is performed to transition from the RV clock domain running at 50 MHz into the faster 250 MHz domain. Because the width of the interface to the PCIe controller is 64-bit, and thus smaller than the bus width monitored, transitioning to the faster clock domain helps to divide the concatenated transaction into manageable sized chunks. The component that takes care of splitting the transactions is the Serializer, and it feeds the 64-bit chunks into another buffer to balance out any delays that the PCIe to host connection might introduce.

3.3 Serializer

At first a split design was considered, consisting of an Arbiter and the Serializer. The Arbiter would first put the cycle counter on a bus wide enough to support any TL channel, and then sequentially any channel that is firing in the current TL transaction. This design was reconsidered, because introducing the TU into a network that supports all channels would mean the arbiter needs six cycles to send the data to the Serializer, with a new transaction possibly coming in every five cycles. This would inevitably lead to a

pipeline overflow during high bandwidth applications. The new Serializer design skips the arbitrating step and splits the channels directly, outputting 64-bit wide chunks. The Serializer also appends a 4-bit channel id field that makes it possible to distinguish the sent transactions on the host side, and adds zero padding to a user specified alignment. The alignment is a minimum of 4, to ensure that later channel ids can be properly identified from the captured data. The alignment can be left out completely when a prefix code is used as channel id. The inner workings of the Serializer is described in greater in section 4.3.1.

3.4 Pipeline Control

The pipeline control contains the cycle counter that increments whenever there is no valid transaction on any channel of the monitored connection and resets to zero when any channel fires. The counter value can then be used by the analyzing application on the host side to identify the exact timings when a transaction was sent. In case the buffers inside the Pipeline are not chosen large enough, and the Pipeline stalls, there are essentially two things that can be done. The first option would be the TU could stall the TL connection that is monitored until the Direct Memory Access (DMA) buffers are free again, and the host is rereading data. This approach would compromise TL's deadlock freedom because a node must not stall for an indefinite amount of time. Even worse, it would stall the data path to the main memory and therefore stall the whole SoC and invalidate any performance benchmarks. This goes against the principle that the TU should act as a passive component to not interfere with the connection monitored. Therefore, it was decided to stop the data capture in case of an overflow, let the pipeline drain empty and then signal the host with an EOF. The status memory is also updated, address one is be set to 1, indicating a pipeline overflow. This way, the application running on the host can decide whether to restart the capture process or keep the already sent valid data. The EOF is also generated when the host disables the Pipeline.

3.5 Control Unit

The control unit acts as an interface to the host pc. It exposes the status memory to the application running on the host side as a way to start and stop the capture process. The status memory also keeps track of events that could happen during the capture process, such as a pipeline overflow or a cycle counter overflow. Because the TU should be extensible in the future, the status memory is placed on the RV side, and write and read requests from the host are first sent through a clock domain crossing, before accessing the memory. The memory itself is 32 bit x 16 words, and holds information about the generated hardware, especially the widths inferred from the TL network. Parts of memory are read only while some of the addresses are writable. Table 3.1 shows the assignment of the status memory registers.

3.6 Data Transfer to Host

The host to capture the traced data is a Linux system, and the Field Programmable Gate Array (FPGA) is the Virtex-7 FPGA VC707 Evaluation Kit running the SoC. Because

Table 3.1: Status Memory Assignment

Address	Function	Writable
0	enable	y
1	overflow	y
2	cycle counter overflow	y
3	cycle counter width	n
4	tl size bits	n
5	tl source bits	n
6	tl sink bits	n
7	tl address bits	n
8	tl data bits	n
9	alignment	n
10	channel id width	n
11	errors/sec	n
12	sent tr/sec	n
13	total errors	n
14	cycle counter	n
15	total tr/sec	

the TU is developed primarily for use within this SoC, an FPGA to host interface must be chosen that is available on the VC707. Furthermore, the interface must also deliver high bandwidth as the MBUS has a throughput of hundreds of megabytes per second. There are several interfaces available, the VC707 supports 1 Gib Ethernet and second generation Peripheral Component Interconnect Express x8. Because of the bandwidth offered by PCIe, this interface was chosen as the option to design the TU. While it would be possible to integrate a Xilinx PCIe IP into the design, it was opted to use a Xillybus IP core[xila], because it offers a simple interface and is therefore easy to use while offering great performance. A significant benefit of Xillybus is its configuration that allows for multiple channels that appear in Linux host as device files. Xillybus also supports a multitude of boards, including the Virtex-7 FPGA VC707 Evaluation Kit. For a smooth data transfer, it is essential to adjust the DMA buffer sizes on the host side to be large enough to buffer events where the application cannot read data for short periods. Xillybus lets the user adjust these buffers and generate multiple streams during the IP generation process. The TU exposes two channels to the host, the primary, high-bandwidth upstream for the data pipeline, and a secondary low-bandwidth seekable device file to access the status memory. On the FPGA side, the device files are exposed as simple native FIFO interfaces.

3.7 Development Process

Designing and verification went hand in hand during the whole development process, and follows a bottom-up approach, create and test small components first before connecting them to form a larger component. Before the start of development, it was first made sure

Chisel, and the PCIe connection is working as intended. After creating several sample projects to check the chisel compilation process and synthesizing the generated Verilog to hardware, the Chisel blackbox feature was tested to make sure the Xillybus [xila] IP is working as intended and data can be sent to the host and back. To test the connection the Xillybus demo bundle [xbd] for the VC707 was imported into chisel as blackbox module. The demo bundle simulates a simple data loop, channeling all data sent from host to FPGA into a FIFO and sending that data back to the host. The demo bundle's FIFO was rewritten in Chisel, and a simple counter was attached. After observing the correct behavior on the host side, it was relied on the correctness of the Xillybus component itself.

This way, the CU was developed and tested, and then the Pipeline. The CU's status memory was checked for correct read- and writability. The Pipeline development was divided into smaller parts. This worked especially well for the Pipeline because the different components are mostly connected with the ready-valid protocol so that each part could be written and tested separately as distinct units. Before joining the TU and CU, both designs were checked if there was data appearing on the host side and in the correct format. In instances where it was not, further functional simulations were done to spot mistakes and correct them. With the same approach, the finished TU was tested, and functional simulation was done when necessary, i.e., when unexpected behavior was observed on the host side.

After observing that the overflow flag was often set rather quickly after starting to capture data, several other performance measuring features were introduced into the CU to analyze unexpected behavior. A transaction counter that measures the total amount of transactions per second happening on the TL bus, and a missed transaction counter that measures how many transactions were missed every second, which happens when a valid transaction could not be inserted into the buffer. Ideally, during normal capture, this will be 0. In the early stages of testing, it occurred that there were a lot of missed transactions. This meant the buffers were overflowing. In conclusion, the Serializer Design was optimized, the Xillybus A core was replaced by a faster XL core that supports higher data rates and more DMA buffer memory on the host side, and the Pipeline's queue sizes were adjusted. With the later introduction of the EOF generation logic, this feature is deprecated because an overflow will end data capture and signal an EOF to the host. Parametrizability increases the testing effort enormously. Because unit tests were not possible, verification for variances in bus widths inferred by Diplomacy was not performed, but only for the widths inferred for the TU being placed directly after the MBUS.

Chapter 4

Implementation Details

This chapter goes in depth about the TU's components and their actual implementations. First the choice of language will be elaborated, followed by detailed descriptions of the components used.

4.1 Choice of Language: Chisel vs VHDL, Verilog

The whole SoC is written in Constructing Hardware in a Scala Embedded Language (Chisel) [Chia], a HDL that upon execution first generates Flexible Internal Representation for RTL (FIRRTL), an intermediate HDL before emitting synthesizable Verilog using the FIRRTL compiler. Chisel promotes high reusability of existing code, resulting in a more compact and concise code for equivalent designs than what is possible with Verilog or VHDL. It supports all the object-oriented, and functional programming paradigms the Scala programming language offers, making it perfect to develop digital circuits with high levels of abstraction. Those benefits are one of the main reasons it is used to develop the U500 platform. U500 is designed to be customizable, and correspondingly, the TL network connecting all the components is lazily parameterized by the Diplomacy network. Because the TU is designed to be placeable anywhere in the TL network, describing it in a traditional HDL, thereby giving up on Diplomacy's parameter inference features, would be suboptimal. A middle ground could be using chisels black box feature to import designs written in Verilog. This way, parameters can still be passed to a parameterizable implementation of the TU, but instantiating the TU as a blackbox would still mean U500's source code has to be modified. Finally, Chisel was chosen to develop the TU, also because of the extensive library of components provided by Chisel and Rocket Chip.

4.2 TileLink Node

The Pipeline, CU and the Xillybus PCIe modules are united under the top TU module, which implements a TileLink Identity Node. The Identity Node allows transactions to be routed through unchanged but exposes the TL channels. The TL bus is routed into the Pipeline, where the data can be processed. CU and Pipeline are connected, as well as both modules are wired up to the Xillybus module.

4.3 Pipeline

All the components in the pipeline are chained together using the valid-ready interface to propagate the data. Because Xillybus expects a native FIFO interface, a conversion was implemented in the last queue. All CDCs in the Pipeline are implemented using Rocket Chip's asynchronous queues. The reset signal for the Pipeline comes from the Xillybus IP. Essentially it is the inverted open signal, which is asserted high when an application accesses the data device file. This reset signal also acts as the reset signal on the RV clock domain. Therefore it has to be first brought to the RV clock domain. Figure 4.1 shows the pipeline implementation.

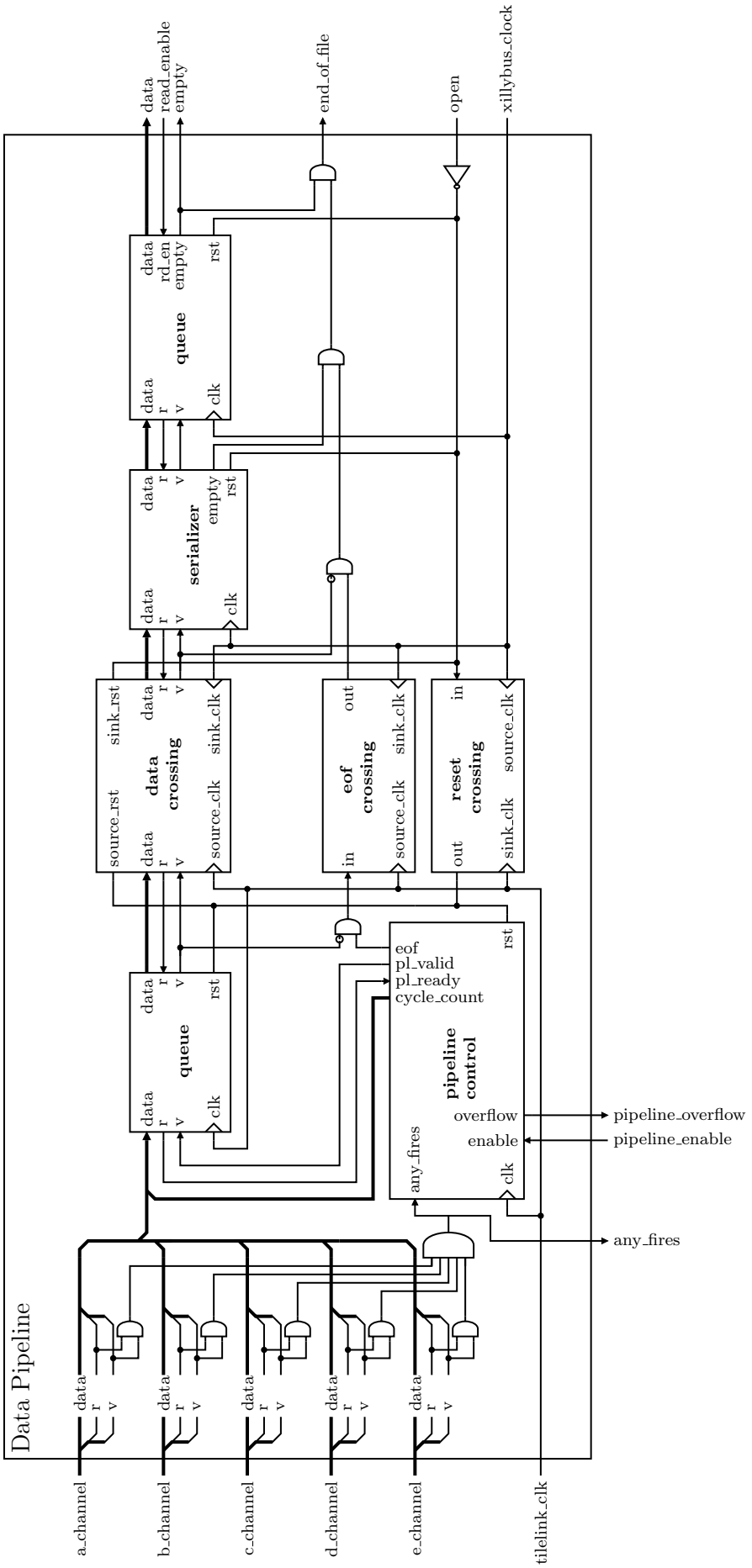

Figure 4.1: Data Pipeline

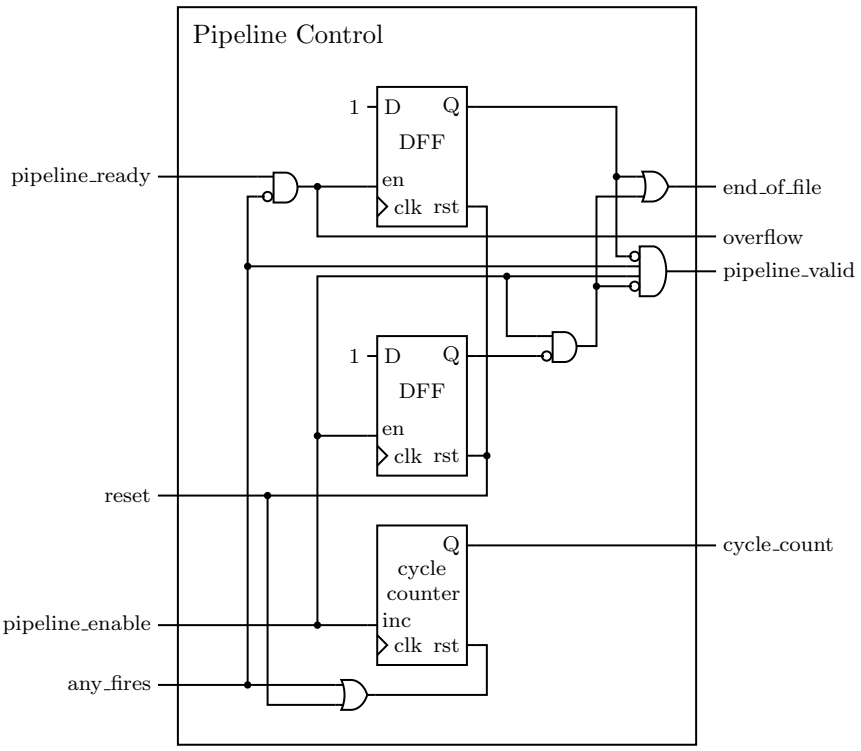

Figure 4.2: Pipeline Control logic inside the Data Pipeline

The EOF is generated in a way that makes sure that the Pipeline is completely emptied before actually letting the EOF go through to the host, meaning all data is sent, and the Pipeline remains empty. When the host disables data capture by setting the status memory address 0 to zero, the EOF is generated after sending out all remaining data in the Pipeline. For the queues, including the async crossing queue, inverting the valid output acts as an empty signal because it is always high when the queue has data. The Serializer has a separate empty output because it can store data for longer while waiting for new incoming transactions. When the TL clock domain's queue is empty, the EOF signal will travel through the crossing. To ensure that data from the data crossing and the EOF signal arrive simultaneously on the Xillybus clock domain, the sync stages for both crossing queues have to be the same. This is ensured by using a global parameter that makes sure that all crossings used in the TU have the same number of sync stages. Figure 4.2 shows the implementation of the pipeline control logic, including the cycle counter.

4.3.1 Serializer

The Serializer first appends each channel with its respective channel id. The resulting channels are then right padded to a multiple of the alignment parameter defined by the user by appending 0s. The padding is necessary to make sure the host script can differentiate between different transactions. Afterwards the permutations of different concatenations of channels are generated, but the respective order is kept, meaning the a-channel is always left of the other channels. Depending on which channels are actually firing the suited concatenation is chosen and afterwards id-extended and padded cycle counter is appended from the left. This way, no matter which channel fires, the cycle counter is always sent out first. Then the data is taken into the serialization stage. It sends data out as 64-bit

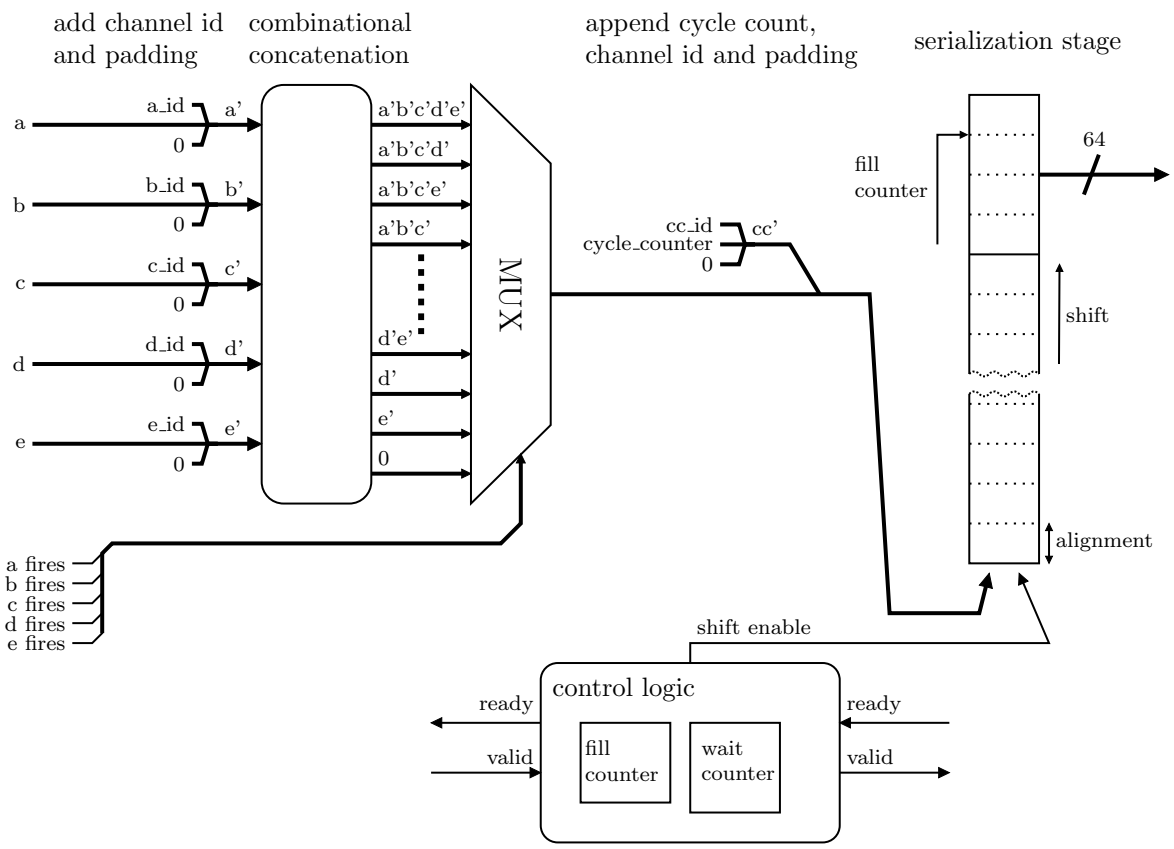

Figure 4.3: Serializer Implementation

chunks to the Xillybus queue. One cycle before the Serializer would be empty, and when the Xillybus queue is ready, the Serializer already assigns ready on its input side. This ensures the that the Serializer can continuously keep firing and no cycle is wasted. In case there is no new data coming from the TL input, the Serializer waits a predefined amount of time before sending out the last bytes. This ensures that no unnecessary zeros are sent to the host. The Serializer's working mechanism is shown in figure 4.3.

4.4 Testing Methods

Occasionally during the development process, it was not completely clear whether the generated hardware resembles what was intended with the chisel source. This was especially true when using more complex functional statements. To quickly check for correctness, small parts of code were compiled and synthesized to inspect the generated Verilog. This method strengthened the feel between writing and knowing what Verilog is generated and helped to understand the Chisel language's quirks, for example, how it handles width inference. Doing this sped up testing significantly because it allows to spot mistakes early, and less time is spent on figuring out bugs during time-consuming simulations. Loading the design on the FPGA and observing the output was usually done before functional simulation. Because the TU collects real data and does not influence the RV processor in any way, the data collected could be observed and cross checked without impacting the rest of the SoC. This way, it was easy to check the design for the CU and the Pipeline for

any undesired behavior, like the data not being sent to the host in the expected format. Functional simulations were performed whenever undesired behavior was apparent. It was also done to verify the Serializer and analyze the behavior of the CDC queues from the Rocket Chip library. There are some resources available on chisel simulation and automated testing with chisel-testers2 [chib], but this testing library is only included in newer versions of Chisel. It turned out that automated testing was not doable with the version used by the U500 Rocket Chip implementation. SiFive must have done testing somehow, but with a lack of documentation on the rocket chip, their testing methods were not clear, and scarce online resources were written for the new test suite. As a result, it was tried to upgrade the chisel version used in the Rocket Chip to a newer release so that the test suite can be used, but that broke several other components inside Rocket, and the changes were reverted. It was therefore resorted to traditional methods, i.e., RTL simulation using Vivado's [Viv] integrated simulator and Tool Command Language (Tcl) scripts.

Chapter 5

Results

After the the integration of the Trace Unit into the SoC the RV was stress tested to analyze its read and write behavior. The following sections will explain the experimental setup before going into detail on how the benchmarks were performed and its results.

5.1 Experimental Setup

The Board used is the Virtex-7 FPGA VC707 Evaluation Kit [Xilb]. It features a Virtex-7 family FPGA, peripherals for serial connectivity with PCIe Gen2x8, UART, and several others. It has a memory interface with 1600Mbps 1GB DDR3 SODIM and features 10-100-1000 Mbps Ethernet. The FPGA itself is the XC7VX485T-2FFG1761, table 5.1 shows the device specifications. Implemented on the FPGA is the Freedom U500. It features four RV GC cores clocked at 50 MHz and running a minimal version of Linux. Xilinx Vivado v2019.2 (64-bit) is used for synthesis and implementation. The resulting resource utilization of the TU is shown in Table 5.2. The host system that the FPGA is connected to has a 64 bit Intel(R) Core(TM) i3-2100 dual-core CPU clocked at 3.10 GHz, running Ubuntu 18.04.4 LTS with 16 GB RAM. The Xillybus Peripheral Component Interconnect Express controller uses the four of the available PCIe lanes and this way the FPGA board is connected to a suitable PCIe x4 slot on the host. The benchmarks are then started by a script to capture and process the data automatically.

5.1.1 Trace Unit Throughput

The memory subsystem from MBUS to DDR in the U500 is implemented using TL-UH, hence only the a- and d-channels are used. Table 5.3 shows the current parametrization of

Table 5.1: Virtex-7 XC7VX485T-2FFG1761 FPGA

Virtex-7:	
Logic Cells	485,760
DSP Slices	2,800
Memory (Kb)	37,080
GTX 12.5 GB/s Transceivers	56
I/O Pins	700

Table 5.2: Trace Unit FPGA resource utilization

Module	Total LUTs	Logic LUTs	LUTRAMs	SRLs	FFs	RAMB36	RAMB18
Trace Unit	9304	8735	502	67	10791	12	4
Control Unit	216	216	0	0	592	0	0
Status Mem	95	95	0	0	378	0	0
Other	121	121	0	0	214	0	0
Pipeline	3301	3297	0	4	3054	4	0
Serializer	850	850	0	0	354	0	0
XB Queue	1680	1680	0	0	246	4	0
Crossing	491	491	0	0	2145	0	0
Other	280	289	0	0	309	0	0
Xillybus	5744	5179	502	63	7145	8	4
Other	43	43	0	4	0	0	0

Table 5.3: Parametrization of the Trace Unit

Bus Widths					Alignment	Queue Depths		CDC	
	Total	Data	Channel Id	Padding		Pre CDC	Xillybus	Depth	Sync Stages
Cycle Counter	32	28*	4*	0					
TileLink					8*	1*	2048*	8*	3*
A Channel	128	121+	4*	3					
D Channel	88	82+	4*	2					

* User specified
+ Inferred from TileLink

the TU. The a-channel is 121 bit wide and the d-channel 82. Appending the 4-bit channel id and padding extends the channels to 16 byte and 11 bytes, respectively. The cycle counter is always 4 bytes wide with 4-bit channel id and 28-bit data fields. With the SoC being clocked at 50 MHz, and both channels in use at every transaction, the monitored connection could reach a theoretical maximum of $50 * 10^6 \frac{1}{s} * (4B + 16B + 11B) = 1550 \frac{MiB}{s}$. This assumes that the Serializer's wait counter is enabled, waiting for the next transaction before sending the previous one out, so that always full 64-bit words are sent to the host. The measurements taken from the benchmarks show that real throughput is much lower, not exceeding 300 MB/s.

5.2 Benchmarks

First the RV's memory connection was measured in idle state, then three benchmarks were executed to test the systems behavior under load. The three benchmarks all consist of simple shell commands and were executed automatically from the host computer. Table 5.4 displays the results of the benchmarks side by side. Key measurements taken were the throughput of reads and writes, as well as memory latency. For reads, the latency measurement is taken from the point where a request was accepted on the a-channel until the first beat of a response burst is accepted. Write latency is taken from the first beat of a write burst until the response acknowledgment is accepted. The high maximum latency can be attributed to the master not being ready to accept a response due to currently accepting data from a different slave on the bus. The data may suggest that the write latency is roughly half the read latency, but this is possibly the result of a node closer to

Table 5.4: Measurements for various benchmarks on the TileLink bus at 50MHz

Benchmark	Idle	Shell Loop 1 Thread	Shell Loop 4 Threads	Fill tmpfs 512 MiB
Duration (s)	30	30	30	76.52
Trace size (MB)	308.516	1710.423	6468.551	10789.143
Throughput (MB/s)	10.283	57.014	215.618	140.997
General Utilization				
Bus Utilization (%)	0.617	3.613	13.451	6.393
Writes				
Transactions (MTr)	1.431	1.649	13.247	178.086
Data (MB)	11.452	13.196	105.977	1424.688
Throughput (MB/s)	0.382	0.439	3.532	18.618
Latency (cycles)				
Min	15	15	15	15
Max	46	47	48	48
Avg	16.677	16.808	16.320	15.132
Reads				
Transactions (MTr)	14.898	94.555	344.168	239.538
Data (MB)	119.186	756.445	2753.351	1916.304
Throughput (MB/s)	3.973	25.215	91.782	25.043
Latency (cycles)				
Min	23	23	23	16
Max	55	53	55	57
Avg	30.573	30.220	30.519	27.133
Per Channel Utilization				
A-channel				
Transactions (MTr)	3.433	13.635	57.694	227.425
Share (%)				
PutFullData	41.620	12.080	22.956	78.302
PutPartialData	0.075	0.0164	0.004	0.002
Get	58.305	87.903	77.040	21.694
Throughput (MTr/s)	0.114	0.454	1.923	2.972
Utilization (%)	0.229	0.909	3.8464	5.944
D-Channel				
Transactions (MTr)	15.077	94.761	345.824	261.799
Share (%)				
AccessAck	1.187	0.217	0.478	8.504
AccessAckData	98.812	99.782	99.522	91.496
Throughput (MTr/s)	0.502	3.158	11.527	3.421
Utilization (%)	1.005	6.317	23.055	6.842

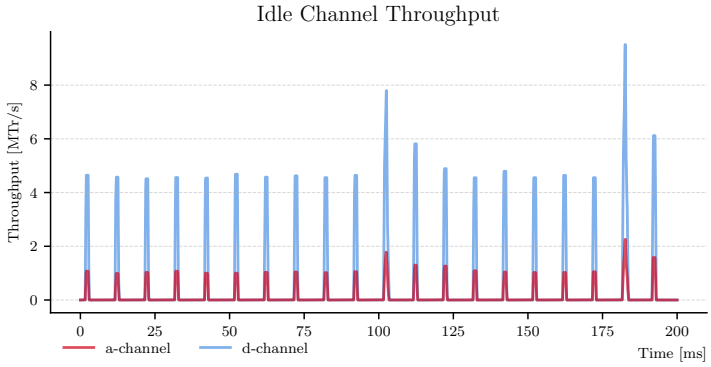

Figure 5.1: RISC-V throughput in idle state on a 200ms scale

the memory controller buffering the data, and therefore sending a write acknowledgment to the master before the data is actually written back to main memory. The TL specification allows that write requests are acknowledged even in the same cycle of the first beat of the write request. Other fields of interest are the utilization of each channel and the distribution of messages sent on the channel. The following sections will describe the benchmarks in greater detail.

5.2.1 Idle State

Figure 5.2 shows the measurements taken of the RV in idle state with only the necessary system processes running. The throughput of the a-channel is averaging at 0.114 M transactions per second, whereas on the d-channel it is about 0.5 M transactions per second. Figure 5.1 shows the throughput of the channels on a 200 ms timescale. It can be seen that the scheduler is invoked every 10 ms, and inbetween the throughput is 0.

5.2.2 Shell Loop with multiple threads

The Shell Loop is a simple test to create multiple processes that execute infinite loops to bring a specified number of cores to 100% utilization. The command used to spawn four processes is:

```
1 for i in 1 2 3 4; do while : ; do : ; done & done
```

Figure 5.4 shows the memory map after spawning four processes. Compared to the idle state there are more pages used, and also the throughput on both channels is significantly higher. Upon inspection of the 200 ms capture, the d-channel throughput is a lot higher than in idle state and never returns to 0. One might assume that only executing a simple while loop should be able to fit inside a cores instruction cache and therefore not creating that many reads. But it appears that is not the case. The 10 ms spikes are still present, indicating the invocation of the scheduler.

5.2.3 Fill Tmpfs 512 MiB

This benchmark creates temporary filesystem in RAM and fills it with 512 MB using the shell command:

```
1 dd if=/dev/zero of=/mnt/tmpfs/test bs=1M count=512
```

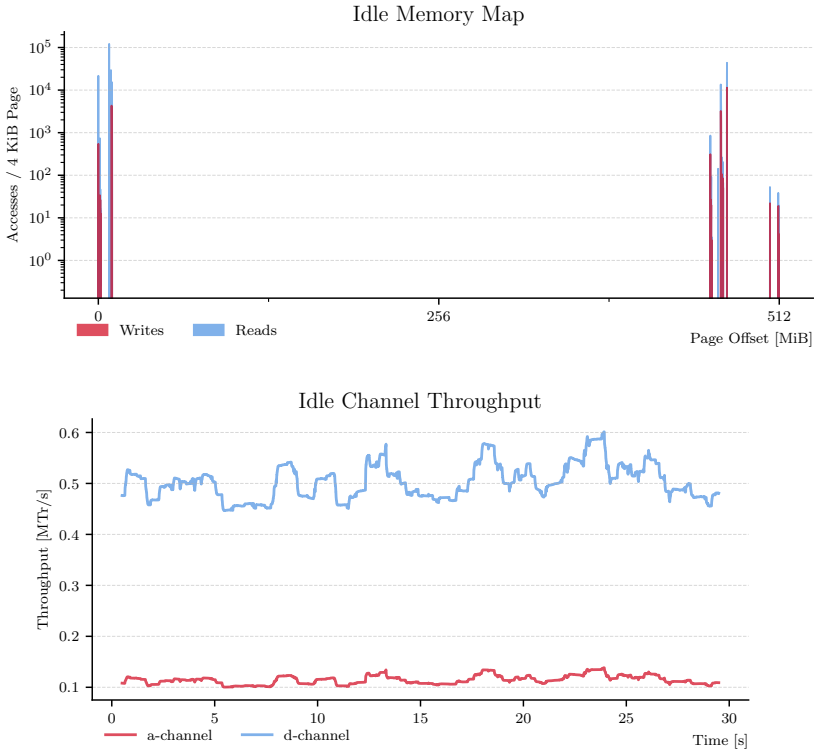

Figure 5.2: RISC-V memory map and throughput in idle state

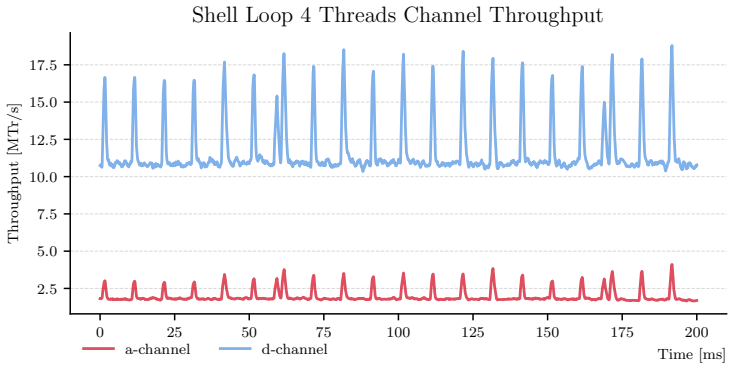

Figure 5.3: RISC-V throughput with Shell loop running 4 threads for 200 ms

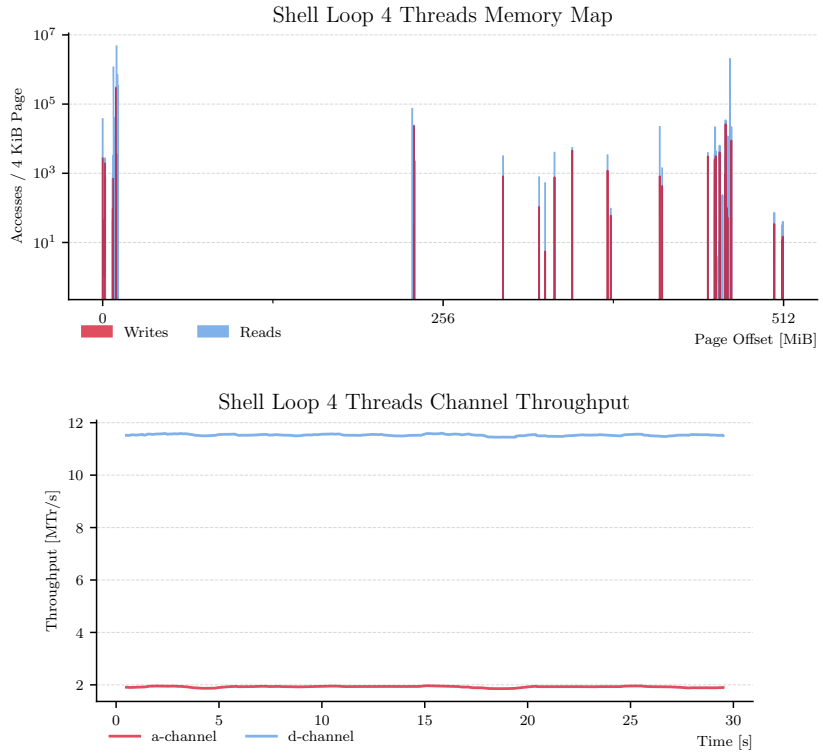

Figure 5.4: RISC-V throughput with Shell loop running 4 threads for 30 s

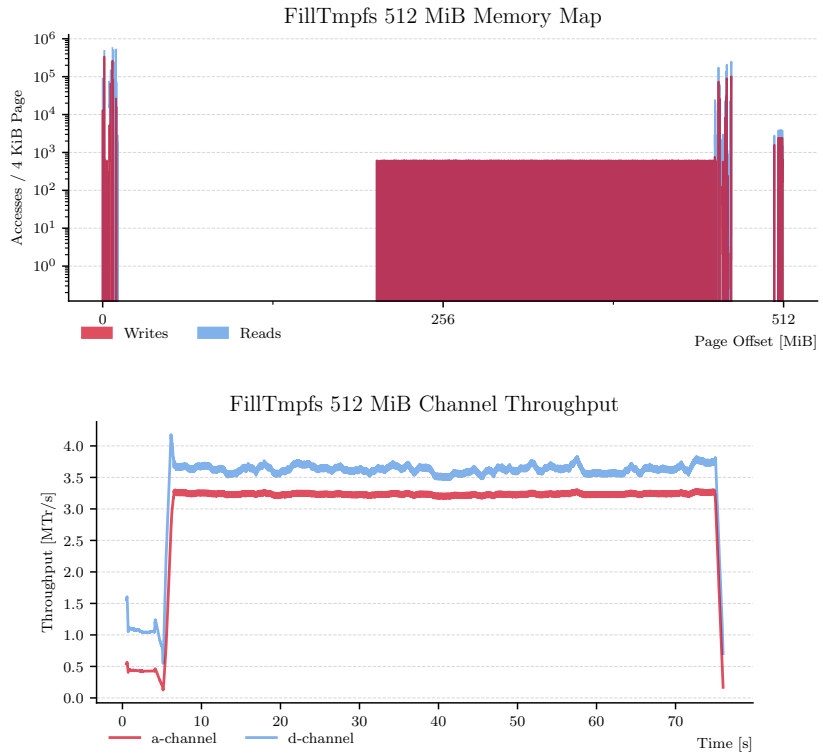

Figure 5.5: RISC-V memory map and throughput reading 512 Mib from /dev/null and writing to a temporary file system

The access map in figure 5.5 shows how the memory was written to in half the address space. The capture process is started slightly before the shell command is executed and stopped soon after it is finished. The throughput graph is showing a substantial increase in throughput on both channels during the writing process.

Chapter 6

Conclusion

6.1 Summary

This work presented the implementation of a Trace Unit that is developed for the Freedom U500 SoC. The steps in the design were shown and the details on the implementation were given. Lastly the results of various benchmarks taken during use of the TU were presented in comparison to the U500's idle state behavior.

6.2 Critics

Due to rapid development of the Rocket Chip Generator (Rocket), the design used in the U500 is already outdated. The components used in the U500 project are partly based on Chisel2, which is an old version of the language and has since been rewritten to be based on Chisel3. Many classes in this version use Chisel3 and Chisel2 interchangeably, making it difficult to understand the code. There are also very limited resources available for the classes used in U500. The chisel error messages not very detailed, needs some digging to figure out what actually went wrong. The U500 implementation also uses a high level of abstraction that is difficult to follow. And lastly the chisel library components are not optimized for a FPGA design flow. For example it is difficult to get the Vivado tools to infer block rams from chisels queue implementation.

6.3 Limitations and Future Work

Currently the TU sends all transactions from the monitored bus to the host. In applications where there is too high throughput on the monitored bus the maximum bandwidth of the PCIe controller might be exceeded. It would be useful to integrate a filter to only record specific TL messages or accesses to predefined memory spaces only. That would allow the capture process to only transmit messages of interest to the host, for example when the TU is connected to monitor a single core, but the user is only interested in writes to a single peripheral that is memory mapped to a smaller address space. The host script is capable to filter address spaces, but a filter implemented in hardware would decrease the amount of data recorded on the host side and therefore speed up analysis significantly. Looking forward it would be useful for measurements to be started and configured from the Linux running on RV itself. That way measurements can be timed more precisely. To

accommodate for this addition the status memory was implemented on RV side. Lastly the correctness for different parameters inferred by Diplomacy when placed into other parts of the TL network was not yet tested, because this requires the integration of automated tests. This is planned for the future.

The host script uses Python records to extract information from the capture file. To be able to extract even more information from the captured data in the future simply extending the script might make it difficult to work with and slow. A new solution using a database would allow for better information extraction using SQL querys.

Bibliography

[AAB+16] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, and Others. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[AMB] Amba – arm developer. https://developer.arm.com/architectures/system-architectures/amba. (Accessed on 09/09/2020).

[Chia] Chisel/firrtl: Home. https://www.chisel-lang.org/. (Accessed on 09/08/2020).

[chib] ucb-bar/chisel-testers2: Repository for chisel3 testers2 open alpha. https://github.com/ucb-bar/chisel-testers2. (Accessed on 09/10/2020).

[CTL17] Henry Cook, Wesley Terpstra, and Yunsup Lee. Diplomatic design patterns: A TileLink case study. In *1st Workshop on Computer Architecture Research with RISC-V*, 2017.

[Git] Github - sifive/freedom: Source files for sifive's freedom platforms. https://github.com/sifive/freedom. (Accessed on 08/28/2020).

[PCI] Pci express gen1/2/3 root fmc module. http://www.hitechglobal.com/FMCModules/FMC_PCIExpress.htm. (Accessed on 08/31/2020).

[SiF16] SiFive. *SiFive Freedom U500 Platform*. SiFive Inc., July 2016.

[SiF19] SiFive. *SiFive TileLink Specification*. SiFive, Inc., August 2019.

[Viv] Vivado design suite. https://www.xilinx.com/products/design-tools/vivado.html. (Accessed on 09/10/2020).

[xbd] Download xillybus for pcie — xillybus.com. http://xillybus.com/pcie-download. (Accessed on 09/10/2020).

[xila] An fpga ip core for easy dma over pcie with windows and linux — xillybus.com. http://xillybus.com/. (Accessed on 09/10/2020).

[Xilb] Xilinx virtex-7 fpga vc707 evaluation kit. https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html. (Accessed on 08/25/2020).