

Design and Implementation of a Scalable IOMMU for RISC-V Architectures

Iason Mastorakis



Thesis submitted in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Prof. *Manolis G.H. Katevenis*

Thesis Supervisor: Dr. *Vassilis D. Papaefstathiou*

This work has been performed at and supported by the Computer Architecture and VLSI Systems (CARV) Laboratory, Institute of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH).

UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

**Design and Implementation of a Scalable IOMMU for RISC-V
Architectures**

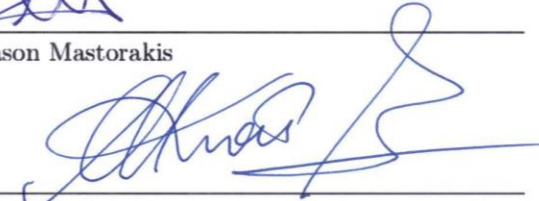
Thesis submitted by
Iason Mastorakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

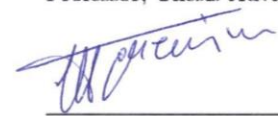
THESIS APPROVAL

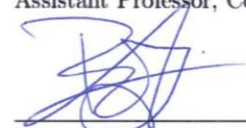
Author:


Iason Mastorakis

Committee approvals:


Manolis GH Katevenis
Professor, Thesis Advisor


Polyvios Pratikakis
Assistant Professor, Committee Member


Vassilis Papaefstathiou
Researcher C, FORTH-ICS, Thesis Supervisor

Departmental approval:


Polyvios Pratikakis
Assistant Professor, Director of Graduate Studies

Heraklion, March 2021

Abbreviations

AR	Address Read
ASID	Address Space Identifier
ATC	Address Translation Controller
ATU	Address Translation Unit
AW	Address Write
AXI	Advanced eXtensible Interface
BRAM	Block Random-Access Memory
CAM	Content Addressable Memory
CDMA	Central Direct Memory Access
CISC	Complex Instruction Set Computer
CPU	Central Process Unit
CTRL	Controller
DMA	Direct Memory Access
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GPU	Graphics Processing Units
HPTW	Hardware Page Table Walker
HW	Hardware
I/O	Input/Output
ID	Identifier

IO-TLB Input/Output Translation Lookaside Buffer

IOMMU Input/Output Memory Management Unit

IPA Intermediate Physical Address

ISA Instruction Set Architecture

LFSR Linear Feedback Shift Register

LRU Least Recently Used

LUT Lookup Table

MM Main Memory

MMU Memory Management Unit

OS Operating System

PA Physical Address

PL Programmable Logic

PPN Physical Page Number

PTBA Page Table Base Address

PTE Page Table Entry

RAM Random-Access Memory

RISC Reduced Instruction Set Computer

RISC-V Reduced Instruction Set Computer Five

RV-IOMMU Reduced Instruction Set Computer Five Input/Output Memory Management Unit

SMH Stream Miss Handler

SMMU System Memory Management Unit

SMU Stream Matching Unit

SRAM Static Random Access Memory

TCM Tightly Coupled Memory

TLB Translation Lookaside Buffer

TMH Translation Miss Handler

VA Virtual Address

VPN Virtual Page Number

Design and Implementation of a Scalable IOMMU for RISC-V Architectures

Abstract

Virtual memory is ubiquitous in general purpose computing systems today because it has many advantages such as simplifying memory management to ease the programmers, offering memory protection and isolation that improves security, and enabling applications to use more memory than the physically available capacity. The virtual memory is managed by the Operating System (OS) and the processors include hardware Translation Lookaside Buffers (TLBs) and Memory Management Units (MMUs) to accelerate virtual-to-physical address translation for the common case. Similarly, I/O devices with Direct Memory Access (DMA) or Graphics Processing Units (GPUs) that do not execute OS code can benefit from virtual memory. For this purpose, many modern architectures offer I/O Virtualization and protection by utilizing specialized Input-Output Memory Management Units (IOMMUs).

This thesis contributes with the hardware design and implementation of an IOMMU for the rising and fast growing open RISC-V architecture ecosystem. We design a scalable IOMMU architecture that supports multiple concurrent I/O devices following the RISC-V specifications for 39- and 48-bit virtual addresses (SV39 and SV48). The design consists of two main components: (a) the Address Translation Unit (ATU) and (b) the Address Translation Controller (ATC). These components are configurable in terms of features and can be combined in several different ways to create scalable and tailored systems with many devices and varying degrees of ATU and ATC sharing. To the best of our knowledge we are among the first to design and implement an IOMMU for RISC-V systems since there are no official specifications published to date (March 2021).

We implement and verify the IOMMU design in SystemVerilog and evaluate its performance using RTL simulation with synthetic traffic patterns that exercise different use cases. Moreover, we evaluate the area and frequency of our IOMMU design on a Xilinx Zynq Ultrascale+ FPGA. Finally, we create an FPGA design that includes our IOMMU and a typical DMA device and we verify its correct functionality on the real system under stress patterns.

Keywords: IOMMU, RISC-V, FPGA, Hardware, Physical Address, Virtual Address

Σχεδίαση και Υλοποίηση μιας Κλιμακώσιμης Μονάδας Διαχείρισης Μνήμης Εισόδου-Εξόδου για Αρχιτεκτονικές RISC-V

Περίληψη

Η εικονική μνήμη είναι πανταχού παρούσα στα συστήματα υπολογιστών γενικού σκοπού επειδή έχει πολλά πλεονεκτήματα όπως την απλοποίηση της διαχείρισης μνήμης για διευκόλυνση των προγραμματιστών, την προστασία μνήμης και την απομόνωση που βελτιώνουν την ασφάλεια, και τη δυνατότητα οι εφαρμογές να χρησιμοποιούν περισσότερη μνήμη από τη διαθέσιμη φυσική μνήμη του συστήματος. Την εικονική μνήμη τη διαχειρίζεται το Λειτουργικό Σύστημα και οι επεξεργαστές περιλαμβάνουν πίνακες μετάφρασης υλοποιημένους σε υλικό (TLBs) και Μονάδες Διαχείρισης Μνήμης (MMUs) για να επιταχύνουν τη διαδικασία μετάφρασης των εικονικών διευθύνσεων σε φυσικές. Αντίστοιχα, συσκευές Εισόδου/Εξόδου με δυνατότητα Άμεσης Προσπέλασης Μνήμης (DMA) και Επεξεργαστές Γραφικών που δεν εκτελούν κώδικα Λειτουργικού Συστήματος μπορούν να επωφεληθούν από τη χρήση εικονικής μνήμης. Για το λόγο αυτό πολλές μοντέρνες αρχιτεκτονικές προσφέρουν Εικονικοποίηση και προστασία για την Είσοδο-Εξοδο χρησιμοποιώντας εξειδικευμένες Μονάδες Διαχείρισης Μνήμης Εισόδου-Εξόδου (IOMMUs) .

Σε αυτή την εργασία σχεδιάστηκε και υλοποιήθηκε μια Μονάδα Διαχείρισης Μνήμης Εισόδου-Εξόδου σε επίπεδο υλικού για το ανερχόμενο και ταχεία αναπτυσσόμενο ανοιχτό οικοσύστημα RISC-V. Σχεδιάσαμε και υλοποιήσαμε μια κλιμακώσιμη αρχιτεκτονική Μονάδας Διαχείρισης Μνήμης Εισόδου-Εξόδου η οποία υποστηρίζει ταυτόχρονα πολλαπλές συσκευές Εισόδου-Εξόδου τηρώντας τις προδιαγραφές της αρχιτεκτονικής RISC-V για εικονικές διευθύνσεις με πλάτος 39 και 48 bits. Η σχεδίαση αποτελείται από δυο κύρια στοιχεία: (α) τη Μονάδα Μετάφρασης Διευθύνσεων (ΜΜΔ) και (β) τον Ελεγκτή Μετάφρασης Διευθύνσεων (ΕΜΔ). Αυτά τα στοιχεία είναι διαμορφώσιμα όσο αφορά τα χαρακτηριστικά τους και μπορούν να συνδυαστούν με πολλούς διαφορετικούς τρόπους έτσι ώστε να δημιουργήσουν επεκτάσιμα συστήματα και να προσαρμοστούν για σχέδια με πολλές συσκευές και διαφορετικούς βαθμούς διαμοιρασμού των ΜΜΔ και ΕΜΔ. Απο όσο είμαστε σε θέση να γνωρίζουμε, είμαστε μεταξύ των πρώτων που σχεδίασαν και υλοποίησαν μια Μονάδα Διαχείρισης Μνήμης Εισόδου-Εξόδου για συστήματα RISC-V καθώς δεν υπάρχουν δημοσιευμένες επίσης προδιαγραφές μέχρι σήμερα (Μάρτιος 2021).

Υλοποιήσαμε και επαληθεύσαμε τη σχεδίαση της Μονάδας Διαχείρισης Μνήμης Εισόδου-Εξόδου σε SystemVerilog και αξιολογήσαμε την απόδοσή της χρησιμοποιώντας προσομοίωση RTL με συνθετικά μοτίβα κυκλοφορίας που εξασκούν διαφορετικά σενάρια χρήσης. Επιπλέον, αξιολογήσαμε τις απαιτήσεις χώρου και τη συχνότητα λειτουργίας της Μονάδας Διαχείρισης Μνήμης Εισόδου-Εξόδου σε μια Xilinx Zynq Ultrascale+ FPGA (συστοιχία επαναπρογραμματιζόμενης λογικής). Τέλος, δημιουργήσαμε ένα σχέδιο σε FPGA που περιέχει τη Μονάδα Διαχείρισης Μνήμης Εισόδου-Εξόδου μας και μια τυπική Μονάδα Άμεσης Προσπέλασης Μνήμης και επαληθεύσαμε τη σωστή λειτουργία σε ένα αληθινό σύστημα κάτω από απαιτητικά μοτίβα δοκιμών.

Acknowledgments

There are so many people that I would like to thank; each one helped me in their unique way.

First of all, I would like to thank my Advisor, Dr. Vassilis Papaefstathiou, for his assistance and guidance throughout my MSc studies. I want to thank him for the continued enthusiasm and willingness to help me at every stage of this thesis. He was always supportive from the beginning of my M.Sc., and he trusted me during my thesis.

Secondly, I would also like to thank my Supervisor, Prof. Manolis GH Katevenis, for his overall assistance and the opportunity that he gave to me to be a member of the CARV's hardware team.

I want to thank Assistant Prof. Polyvios Pratikakis for being a member of my M.Sc. Committee and his feedback.

I would also like to thank all the “hardware guys” of the CARV that helped me evaluate my implementation on an FPGA. Specifically, I want to thank Nikolaos Dimou, Michalis Giaourtas, Giorgos Ieronimakis, and Aggelos Ioannou.

Moreover, I would like to thank my fellow student, a good friend and “partner in crime,” Sotiris Totomis.

I need to express my gratitude to the University of Crete and the Department of Computer Science, and the Institute of Computer Science of the Foundation for Research and Technology for supporting me.

στους γονείς μου

Contents

Table of Contents	i
List of Tables	iii
List of Figures	v
1 Introduction	1
1.1 Contributions	2
2 Background and Related Work	5
2.1 Virtual Memory	5
2.1.1 Comparison between Physical and Virtual Addressing . . .	5
2.2 RISC-V	6
2.2.1 Page Tables on RV64	7
2.2.2 Address Translation Process on RV64	9
2.2.2.1 Differences between SV39 and SV48	10
2.3 Advanced Extensible Interface 4	11
2.3.1 Introduction	11
2.3.2 Channels	11
2.3.3 AXI Read Transactions	14
2.3.4 AXI Write Transactions	15
2.3.5 Requirements	15
2.4 Related Work	15
2.4.1 ARM IOMMU (System MMU)	15
2.4.1.1 The Stream mapping table	16
2.4.1.2 Translation Context	16
3 Design and Implementation	19
3.1 Overview	19
3.2 Design Overview	20
3.3 Address Translation Unit	24
3.3.1 Stream Matching Unit	25
3.3.1.1 (Optional) AR-FIFO & AW-FIFO	26
3.3.1.2 Uniform AXI struct	26

3.3.1.3	Input CTRL	27
3.3.1.4	AXIID to ASID Set Associative Cache	27
3.3.1.5	Unique entries FIFO	28
3.3.1.6	Replay (Duplicate) FIFO	28
3.3.1.7	Registers that hold data transmitted to the ATC .	29
3.3.1.8	Response Registers	29
3.3.1.9	Output CTRL	29
3.3.2	Input Output Translation Look-aside Buffer (IO-TLB) . . .	29
3.3.2.1	Input CTRL	30
3.3.2.2	Translation Lookaside Buffer (Sv39/Sv48)	30
3.3.2.3	Unique entries FIFO	31
3.3.2.4	Replay (Duplicate) FIFO	31
3.3.2.5	Registers that hold data transmitted to the ATC .	31
3.3.2.6	(Optional) Pipeline Register (Sv39/Sv48)	32
3.3.2.7	Construct Physical Address & Permission Checker (Sv39/Sv48)	32
3.3.2.8	Re-construct AXI struct	32
3.3.2.9	(Optional ROB) AXI AW FIFO	32
3.3.2.10	(Optional ROB) AXI AR FIFO	33
3.3.3	Controller Unit	33
3.4	Address Translation Controller	33
3.4.1	ATC Controller	34
3.4.2	Stream Miss Handler	35
3.4.3	Translation Miss Handler	35
3.4.4	FIFOs	35
3.5	Communication between Address Translation Unit and Address Trans- lation Controller	36
3.5.1	Request switch	36
3.5.2	Response switch	36
4	Evaluation	37
4.1	Phase 1: Simulation	37
4.1.1	Metrics	38
4.1.1.1	The performance cost of reordering	39
4.1.1.2	Performance sensitivity analysis on ATU's TLB size	50
4.1.1.3	Performance sensitivity analysis on SMU's FIFO size	54
4.1.1.4	Performance sensitivity analysis on TLB's FIFO size	56
4.1.1.5	Trying to simulate an actual system under a burst	58
4.2	Phase 2: Evaluation on FPGA	63
4.2.1	Target Platform	63
4.2.2	Implemented Experiment	63
4.2.3	Timing Requirements	64
4.2.4	Synthesis Utilization	66
4.2.5	Implementation Utilization	68

5	Conclusion and future work	71
5.1	Summary	71
5.2	Future Work	71
	Bibliography	71
A	ARM’s Compressed StreamID indexing matching algorithm	75

List of Tables

2.1	Comparison between RISC-V SV39 & SV48 supported page sizes .	11
2.2	Signals of AXI's Write Address (AW) channel	13
2.3	Signals of AXI's Read Address (AR) channel	14
4.1	RV-IOMMU configurations used to explore the cost of the reordering feature feed by a randomized traffic generator	40
4.2	Time (in cc) results of configurations of Table 4.1 used to explore the cost of the reordering feature (randomized traffic generator) . .	41
4.3	Architectural metrics of configurations of Table 4.1 used to explore the cost of the reordering feature (randomized traffic generator) . .	41
4.4	RV-IOMMU configurations used to explore the cost of the reordering feature feed by a sequential traffic generator	46
4.5	Time (in cc) results of configurations found in Table 4.4, used to explore the cost of the reordering feature (sequential traffic generator)	47
4.6	Architectural metrics of configurations of Table 4.1 used to explore the cost of the reordering feature (sequential traffic generator) . . .	48
4.7	RV-IOMMU configurations used to explore the performance effects of ATU's L1 TLB size (random traffic generator)	51
4.8	Time (in cc) results of configurations of Table 4.7, used to explore the performance effects of ATU's L1 TLB size (random traffic generator)	51
4.9	Architectural measurements of configurations of Table 4.7 exploring the performance effects of the ATU's L1 TLB size (random traffic generator)	52
4.10	RV-IOMMU configurations used to explore the performance effects of ATU's L1 TLB size (sequential traffic generator)	53
4.11	Time (in cc) results of configurations of Table 4.10, used to explore the performance effects of ATU's L1 TLB size (sequential traffic generator)	53
4.12	Architectural results of configurations of Table 4.10 exploring the performance effects of the ATU's L1 TLB size (sequential traffic generator)	54
4.13	RV-IOMMU configurations used to explore the performance effects of the SMU's FIFO sizes (random traffic generator)	55

4.14	Time (in cc) results of configurations of Table 4.13, used to explore the performance effects of the SMU's FIFO size (random traffic generator)	55
4.15	Architectural measurements of configurations of Table 4.13 exploring the performance effects of the SMU's FIFOs size (random traffic generator)	56
4.16	RV-IOMMU configurations used to explore the performance effects of the TLB's FIFO sizes (random traffic generator)	57
4.17	Time (in cc) results of configurations of Table 4.16, used to explore the performance effects of the TLB's FIFO size (random traffic generator)	57
4.18	Architectural measurements of configurations of Table 4.16 exploring the performance effects of the SMU's FIFOs size (random traffic generator)	58
4.19	RV-IOMMU configurations used to explore the performance of RV-IOMMU for multiple numbers of connected ATUs feed by sequential traffic generator	60
4.20	Table's 4.19 time (in cc) results of configurations. These configurations were used to explore the RV-IOMMU performance when multiple ATUs are connected and fed by the sequential traffic generator.	61
4.21	Architectural metrics of configurations of Table 4.19 used to explore the performance of RV-IOMMU for multiple numbers of connected ATUs feed by a sequential traffic generator	62
4.22	RV-IOMMU's timing requirements for different configurations comparing the effect of input buffers of ATU	65
4.23	HW resources for RV-IOMMU with one ATU and one ATC	66
4.24	HW resources for RV-IOMMU with one ATU and one ATC with ATU's reordering feature	67
4.25	HW resources for RV-IOMMU with one ATU and one ATC with ATU's reordering feature and input buffers	67
4.26	Actual HW resources for RV-IOMMU with two ATU and one ATC implemented on the FPGA	69
4.27	Actual HW resources comparison for (a) RV-IOMMU with two ATU and one ATC, (b) Ariane core, and (c) CDMA - all implemented on the same FPGA	69

List of Figures

2.1	A system that uses physical addressing	6
2.2	A system that uses virtual addressing	6
2.3	SV39 page table entry Source: The RISC-V Instruction Set Manual Volume II: Privileged Architecture [10]	8
2.4	SV48 page table entry Source: The RISC-V Instruction Set Manual Volume II: Privileged Architecture [10]	8
2.5	SV39 Virtual Address Source: The RISC-V Instruction Set Manual Volume II: Privileged Architecture [10]	8
2.6	SV39 Physical Address Source: The RISC-V Instruction Set Manual Volume II: Privileged Architecture [10]	8
2.7	SV48 Virtual Address Source: The RISC-V Instruction Set Manual Volume II: Privileged Architecture [10]	8
2.8	SV48 Physical Address Source: The RISC-V Instruction Set Manual Volume II: Privileged Architecture [10]	9
2.9	Abstract address translation with multi-level page tables	10
2.10	AXI channels	12
3.1	An example of where an IOMMU could be located in a system. . .	20
3.2	Examples of where an SMMU could be located in a system. Source: ARM	20
3.3	Abstract schematic of RV-IOMMU instantiation with a unique Ad- dress Translation Controller	22
3.4	Abstract schematic of RV-IOMMU instantiation with many Address Translation Controllers	22
3.5	RV-IOMMU Overview	23
3.6	Address Translation Unit Overview	25
3.7	Address Translation Unit Timing Diagram	25
3.8	Stream Matching Unit	26

3.9	AXI ID to ASID Cache Architecture	28
3.10	Input/Output Translation Lookaside Buffer	30
3.11	Address Translation Controller Overview	34
3.12	Address Translation Controller Timing Diagram	34
4.1	Abstract schematic of the simulation environment	38
4.2	Histogram of clock cycles that RV-IOMMU requires to accomplish the run on Conf. 1 of Table 4.1	42
4.3	Histogram of clock cycles that RV-IOMMU requires to accomplish the run on Conf. 2 of Table 4.1	43
4.4	Histogram of clock cycles that RV-IOMMU requires to accomplish the run on Conf. 3 of Table 4.1	43
4.5	Histogram of clock cycles that RV-IOMMU requires to accomplish the run on Conf. 4 of Table 4.1	44
4.6	Histogram of clock cycles that RV-IOMMU requires to accomplish the run on Conf. 5 of Table 4.1	44
4.7	Histogram of clock cycles that RV-IOMMU requires to accomplish the run on Conf. 6 of Table 4.1	45
4.8	Histogram of clock cycles that RV-IOMMU requires to accomplish the run on Conf. 1 of Table 4.4	46
4.9	Histogram of clock cycles that RV-IOMMU requires to accomplish the run on Conf. 2 of Table 4.4	47
4.10	Histogram of clock cycles that RV-IOMMU requires to accomplish the run on Conf. 3 of Table 4.4	48
4.11	Histogram of clock cycles that RV-IOMMU requires to accomplish the run on Conf. 4 of Table 4.4	49
4.12	Histogram of clock cycles that RV-IOMMU requires to accomplish the run on Conf. 5 of Table 4.4	49
4.13	Histogram of clock cycles that RV-IOMMU requires to accomplish the run on Conf. 6 of Table 4.4	50
4.14	Implemented block design of FPGA evaluation	64
4.15	Critical path with source the ATU's AXI ID to ASID Cache and destination the Unique Entries TLB's FIFO	65

Chapter 1

Introduction

In today's computer systems and particularly the more resource-intensive ones, like servers, I/O transactions, such as reading data from a hard disk or a network card, constitute a significant part of the overall workload execution time, making them an important part for system performance. Most of today's peripheral devices bypass the processor to minimize the usage of the OS Kernel [8] and read and write directly from and to the Main Memory (MM) using the Direct Memory Access (DMA) hardware unit [14], [1].

While directly accessing the memory comes with performance benefits, it also leads to stability and protection issues when, for instance, an external device attempts to access a chunk of memory that it should not. An example is when a peripheral device writes to an address that is part of a system's running process which in the best-case scenario will cause the process to terminate. However, one should consider the possible dangers when an external device attempts to read some sensitive data and forward them to the network. In the latter case, the results could be catastrophic. For instance, an external device with direct memory access could be programmed to erase a chunk of the system's memory or trigger various system bugs. Moreover, it can lead to malware redistribution across different systems [15].

Such issues led to creating a mechanism that applies the Virtual Memory to the CPU. This mechanism provides a solution to this problem by introducing an abstraction layer that prevents direct access to physical memory. Computer architects and designers use this mechanism to resolve the CPU-related issues and subsequently started utilizing similar mechanisms for I/O devices (called to I/O virtual memory). The I/O Memory Management Unit (IOMMU) is the hardware unit responsible for the I/O virtual memory, which translates virtual addresses that are peripheral-visible to physical ones, unlike the MMU, which translates CPU-visible virtual addresses to physical.

Over the past decade, both Intel and AMD include IOMMUs to their chipsets, making them available the mainstream computer systems. Since then, multiple processor architectures have followed this path. For instance, ARM implements

an IOMMU-alike mechanism called SMMU [5]. Nowadays, it is clear that the IOMMU's benefits are of high importance for the world of computer architecture. Some of the advantages of having an IOMMU instead of using direct physical addressing of the memory include: (a) the ability to allocate large regions of memory without the need to be contiguous in physical memory as the IOMMU maps contiguous virtual addresses to the underlying fragmented physical addresses, (b) devices that do not support memory addresses long enough to address the entire physical memory can still address the entire memory through the IOMMU, avoiding overheads associated with copying buffers to and from the peripheral's addressable memory space, (c) memory is protected from malicious devices attempting to access a chunk of memory that is not allowed, (d) in systems that run hypervisors and use virtualization, guest operating systems can utilize explicitly the hardware that supports I/O virtualization. High performance hardware such as graphics cards use DMA to access memory directly; in a virtual environment, all memory addresses are re-mapped by the virtual machine software, which causes DMA devices to fail. The IOMMUs can handle such re-mappings, allowing the native device drivers to be used in a guest operating system [7].

This thesis aims to design and implement a scalable IOMMU compatible with RISC-V architectures [17]. To this end, we design and implement an RV-IOMMU that contains: (a) one or more Address Translation Unit(s) connected with one AXI channel that performs address translation at the appropriate busses, (b) an ATC that handles ATU misses by communicating with the main memory – through an AXI-Lite protocol – to read Page Tables and the AXIID2ASID Table (see Section 3 for more details) and (c) custom interconnect switches responsible for connecting one or more ATUs to one ATC. The scalability of the implementation is a result of allowing the customization of the number of ATUs and ATCs. We achieve this by introducing a simple handshake action among the involved components. Lastly, the RV-IOMMU we implement in this work supports the 64-bit RISC-V architecture, including the SV39 and SV48 as described in the Privileged Architecture of the RISC-V Instruction Set Manual [10].

The rest of this thesis is organized as follows: Chapter 2 provides the background and related work, Chapter 3 presents the design of the implementation of our IO-MMU, Chapter 4 describes the experimental evaluation and metrics. Finally, in Chapter 5, we offer our conclusions and provide directions for future work.

1.1 Contributions

The author of this thesis designed and built a hardware implementation of the RV-IOMMU. This implementation was created and tested using SystemVerilog [2], a hardware description and verification language used to model, design, simulate, test, and implement electronic systems. It is crucial to mention that almost everything was designed and built by the author of this thesis. However, the author

used as reference the implementations of: (a) the Translation Lookaside Buffer (TLB) and (b) the Hardware Page Table Walker (HPTW) from the Ariane core, an open-source 64-bit RISC-V Application-Class Processor [6]. The latter two modules could not be used as is from the Ariane implementation since they have structures and interfaces that are tightly coupled with the core pipeline and require substantial modifications. For instance the HPTW interfaces in the author's implementation use the internal interfaces of the ATC and an external AXI-Lite protocol interface that allows accessing main memory and offers compatibility with a plethora of memory blocks that use AXI interfaces, unlike Ariane's custom interfaces. For case of the TLB implementation, the author developed custom logic regarding the replacement policy to enhance performance. Lastly, the implementation of these two modules in Ariane follows the SV39 specification, whereas, the author also implemented the SV48 specification in addition to SV39.

For the evaluation phase the author used an RTL simulator to produce metrics related to performance and validation of the RV-IOMMU implementation. Moreover, the author created an FPGA design to test the implementation of the RV-IOMMU on the Programmable Logic of Zynq Ultrascale+ Xilinx MPSoC FPGA. The FPGA design contains: (a) the RV-IOMMU, (b) a CDMA Unit, and (c) AXI interconnects for the internal communication. For the tests on the FPGA design the author developed C-based bare-metal software that executes on a core of the Zynq SoC. See Section 4 for more details.

Chapter 2

Background and Related Work

2.1 Virtual Memory

The first system implemented using the technique of virtual memory was a one-level storage system in the Atlas Computer [16] in 1962. In this system, a paging mechanism was used to map the programmer's usable virtual addresses to the actual memory. Although Atlas was the first computer that used this technique, it was Fritz-Rudolf Güntsch from the Technische Universität Berlin who first introduced the concept of virtual memory in his doctoral thesis, namely Logical Design of a Digital Computer with Multiple Asynchronous Rotating Drums and Automatic High-Speed Memory Operation, in 1956 [13]. The virtual memory provides a clean and practical programming model to the user, and it is an idealized abstraction of the storage resources available on a given machine. Programs perform memory accesses using only virtual addresses, and the CPU and OS work together to translate those virtual addresses into physical ones that specify the physical location of the data. Some of the main advantages of using virtual memory is that: (a) provide to programmers a large uniform address space which allows them to focus on designing the program rather than dealing with managing the memory used by it (b) it makes the code portable from one machine to another, (c) create the illusion to the programmer that an allocated large region of memory is continuous (d) provides protection and isolation of each process's address space from corruption by other processes makes multi-programming easy and secure to use and (e) more efficient utilization of the main memory (MM) by treating it as a cache for process data stored on disk, keeping only the active areas in MM, and swapping data disk and memory as needed.

2.1.1 Comparison between Physical and Virtual Addressing

This subsection compares a machine's operation using physical addresses to the operation using virtual addresses. This comparison would allow the user to understand how these two approaches are being used from a machine. In Figure 2.1, the CPU produces a physical address to load/store the data from MM. This address

is the absolute position of the word that the CPU wants to read/write in the data array of the MM. As depicted in the example, to access the word starting from address 4, the addresses that the CPU wants to access are 4,5,6, and 7.

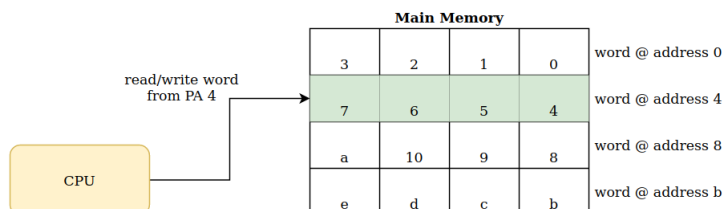


Figure 2.1: A system that uses physical addressing

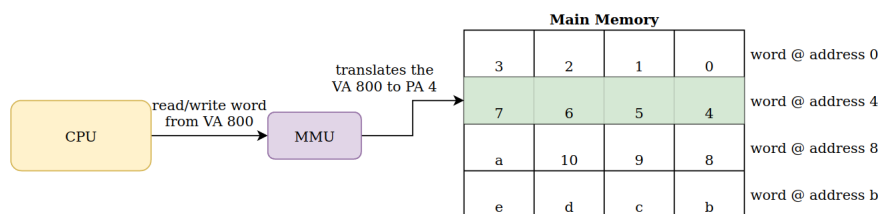


Figure 2.2: A system that uses virtual addressing

On the other hand, Figure 2.2 describes a system that the CPU generates a virtual address, and the MMU translates it to a physical one to get all the benefits of using the concept of Virtual memory. In general, the MMU maps virtual space pieces to equal-sized pieces of physical space. In this way, when the CPU reads from or writes to a memory location, it indicates the virtual space numbering location, i.e., the virtual address. The MMU is a hardware component that intervenes and converts that address to a physical location in real memory (i.e., the physical address). More specifically, Figure 2.2 presents a case where the CPU wants to access the word at virtual address 800, and the MMU translates this address on physical address 4. To translate the virtual address onto the physical one, the MMU accesses the Page Tables by applying an algorithm described in subsection 2.2.2, and it depends on the system's ISA.

2.2 RISC-V

The concepts of modern Reduced Instruction Set Computers (RISC) date back to the 1980s. In short, the main idea is to make the hardware simpler by having a reduced number of simple instructions for loading, evaluating, and storing data. In contrast, the Complex Instruction Set Computer (CISC) architecture utilizes complex instructions that typically include the load and store steps. Both ideas

aim to make CPUs faster – RISC by reducing the cycles per instructions at the cost of the overall number of instructions per application, and CISC by minimizing the number of instructions per program at the cost of more complex hardware. RISC-V Instruction Set Architecture (ISA) is open-source and free for personal, academic and commercial use standard. ISA is the interface between hardware and software. To be useful, ISA demands support from both sides. The RISC-V ISA's notable features include (a) a load-store architecture, (b) bit patterns to simplify the multiplexers in a CPU, (c) IEEE 754 floating-point, (d) an architecturally neutral design, (e) and placing most-significant bits at a fixed location to speed sign extension. The instruction set is designed for a wide range of uses. The base instruction set has a fixed length of 32-bit naturally aligned instructions, and the ISA supports variable-length extensions where each instruction could be any number of 16-bit parcels in length. The instruction set specification defines 32-bit and 64-bit variants.

Ongoing efforts from both the research and industry areas aim to replace the power-hungry, high-end servers with simpler, RISC-like ones, coupled with accelerators and DMAs, to reduce the energy consumption and the overall system cost. That has a significant impact on our choice to develop an IOMMU compatible with RISC-V architectures compatible with 64-bit virtual address space. RISC-V 64-bit is supporting both SV39 and SV48. The difference between the SV39 and SV48 is the Virtual Address Space, where SV39 supports 39-bit virtual address space and SV48 supports 48-bit virtual address space. Address space is a range of valid and discrete addresses, each of which may correspond to a network host, peripheral device, disk sector, a memory cell, or other logical or physical entity. The address space size is the number of bits needed to represent the largest address space. For example, a virtual address space with $N = 2^N$ addresses is called an N-bit virtual address space.

2.2.1 Page Tables on RV64

A Page Table is a data structure generated by the OS which contains 2^9 Page Table Entries (PTEs) stored on the MM of the system. A PTE follows RISC-V ISA's structure, which is this subsection's main point of focus. More details can be found in RISC-V's privileged manual [10]. Each PTE holds a mapping between a page's virtual address and a physical frame address. Figure 2.3 describes a PTE format according to the SV39 of RV64 ISA, whereas Figure 2.4 shows the format when SV48 is enabled. The main difference is that in the SV48 structure, there are four fields of Physical Page Number (PPN), while in the SV39 one, only three of them exist. In both cases, the total width of all PPNs can be up-to 44-bits. Consequently, when SV48 is enabled, an extra translation level is added.

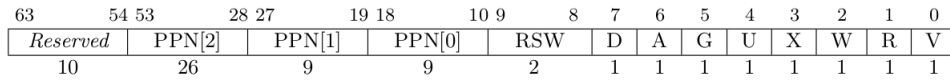


Figure 2.3: SV39 page table entry

Source: The RISC-V Instruction Set Manual Volume II: Privileged Architecture [10]

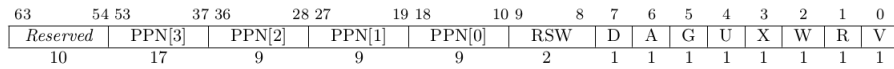


Figure 2.4: SV48 page table entry

Source: The RISC-V Instruction Set Manual Volume II: Privileged Architecture [10]

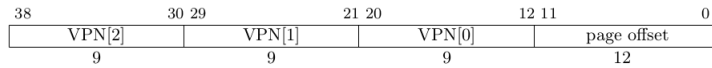


Figure 2.5: SV39 Virtual Address

Source: The RISC-V Instruction Set Manual Volume II: Privileged Architecture [10]

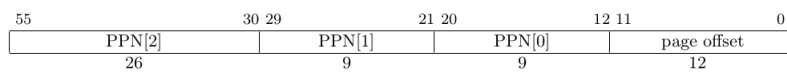


Figure 2.6: SV39 Physical Address

Source: The RISC-V Instruction Set Manual Volume II: Privileged Architecture [10]

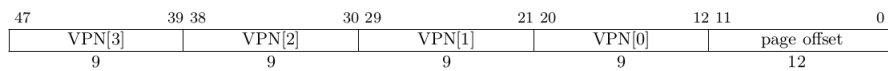


Figure 2.7: SV48 Virtual Address

Source: The RISC-V Instruction Set Manual Volume II: Privileged Architecture [10]

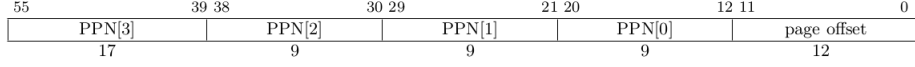


Figure 2.8: SV48 Physical Address

Source: The RISC-V Instruction Set Manual Volume II: Privileged Architecture [10]

2.2.2 Address Translation Process on RV64

This subsection presents an overview of a virtual address translation process to a physical one. More details can be found in the RISC-V Privileged Manual [10]. Firstly, Figure 2.5 presents the VA of Sv39 that is translated on the PA of Figure 2.6. The same applies to Sv48, described in Figure 2.7 and Figure 2.8, respectively. The process of translation of a virtual address into a physical one as defined in the privileged specification is:

1. Let a be $\text{satp.ppn} \times \text{PAGESIZE}$, and let $i = \text{LEVELS} - 1$. Where the physical page number of the root page table is stored in the satp register's PPN field (For Sv39, $\text{PAGESIZE} = 2^{12}$ and $\text{LEVELS}=3$ and for Sv48, $\text{PAGESIZE} = 2^{12}$ and $\text{LEVELS}=4$.)
2. Let pte be the value of the PTE at address $a + \text{va.vpn}[i] \times \text{PTESIZE}$. (For Sv39, $\text{PTESIZE}=8$ and for Sv48 $\text{PTESIZE}=8$.) If accessing pte violates a PMA or PMP check, raise an access exception corresponding to the original access type.
3. If $\text{pte.v} = 0$, or if $\text{pte.r} = 0$ and $\text{pte.w} = 1$, stop and raise a page-fault exception corresponding to the original access type.
4. Otherwise, the PTE is valid. If $\text{pte.r} = 1$ or $\text{pte.x} = 1$, go to step 5. Otherwise, this PTE is a pointer to the next level of the page table. Let $i = i - 1$. If $i \nless 0$, stop and raise a page-fault exception corresponding to the original access type. Otherwise, let $a = \text{pte.ppn} \times \text{PAGESIZE}$ and go to step 2.
5. A leaf PTE has been found. Determine if the requested memory access is allowed by the pte.r , pte.w , pte.x , and pte.u bits, given the current privilege mode and the value of the SUM and MXR fields of the mstatus register. If not, stop and raise a page-fault exception corresponding to the original access type.
6. If $i > 0$ and $\text{pte.ppn}[i - 1 : 0] \neq 0$, this is a misaligned superpage; stop and raise a page-fault exception corresponding to the original access type.
7. If $\text{pte.a} = 0$, or if the memory access is a store and $\text{pte.d} = 0$, either raise a page-fault exception corresponding to the original access type, or:

- Set $pte.a$ to 1 and, if the memory access is a store, also set $pte.d$ to 1.
- If this access violates a PMA or PMP check, raise an access exception corresponding to the original access type.
- This update and the loading of pte in step 2 must be atomic; in particular, no intervening store to the PTE may be perceived to have occurred in-between.

8. The translation is successful. The translated physical address is given as follows:

- $pa.pgoff = va.pgoff$.
- If $i > 0$, then this is a superpage translation and $pa.ppn[i - 1 : 0] = va.vpn[i - 1 : 0]$.
- $pa.ppn[LEVELS - 1 : i] = pte.ppn[LEVELS - 1 : i]$.

2.2.2.1 Differences between SV39 and SV48

The specification for the SV39 defines three different levels of the size of the pages, whereas the SV48 defines four. As presented in Table 2.1, SV39 supports 1GiB Gigapages, 2MiB megapages, and 4KiB pages, whereas SV48 supports all of them plus 512GiB terapages. That occurs because any PTE could be a leaf PTE. Moreover, all pages must be virtually and physically aligned to a boundary equal to their size. Page tables contain 2^9 PTEs, eight bytes each. In SV39, the 27-bit VPN is translated into a 44-bit PPN via a three-level page table, while the 12-bit page offset remains untranslated. In SV48, the 36-bit VPN is translated into a 44-bit PPN via a four-level page table, while the 12-bit page offset remains – once again – untranslated. Figure 2.9 presents an abstract schematic of the translation process on a multi-level page table hierarchy like RISC-V follows.

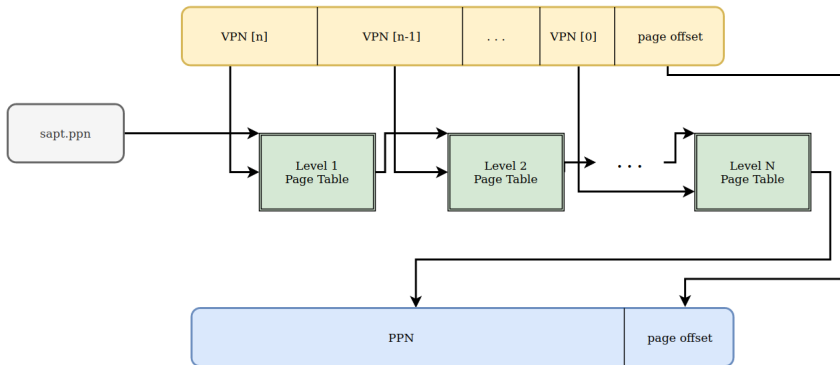


Figure 2.9: Abstract address translation with multi-level page tables

RISC-V SV39 & SV48 supported page sizes		
Page Size	SV39	SV48
4KiB (Kilopages)	✓	✓
2MiB (Megapages)	✓	✓
1GiB (Gigapages)	✓	✓
512GiB (Terapages)	-	✓

Table 2.1: Comparison between RISC-V SV39 & SV48 supported page sizes

2.3 Advanced Extensible Interface 4

2.3.1 Introduction

The Advanced eXtensible Interface (AXI) [4], is an interface protocol defined by ARM as part of the AMBA (Advanced Microcontroller Bus Architecture) standard [3]. The AXI specification describes a point-to-point protocol between two interfaces: a master and a slave. There are three defined types of the AXI-4 Interfaces:

- AXI4 (Full AXI4): For high-performance memory-mapped requirements.
- AXI4-Lite: For simple, low-throughput memory-mapped communication (for example, to and from control and status registers).
- AXI4-Stream: For high-speed streaming data.

2.3.2 Channels

A channel is an independent collection of AXI signals associated with the VALID and READY signals. To increase the interface's bandwidth, AXI uses separate address and data channels for read and write transfers. There is no timing relationship between the groups of read and write channels. This means that a read sequence can happen simultaneously as a write one. The AXI protocol defines 5 channels that share the same handshake mechanism based on the VALID and READY signals. The VALID signal is assigned from the source to the destination, whereas the READY is being assigned vice-versa. A transfer happens when both the VALID and READY signals are high while there is a clock's rising edge. These 5 channels are:

- Write Address (AW)
- Write Data (W)
- Write Response (B)
- Read Address (AR)

- Read Data (R)

While performing a simple handshake between master and slave, the address channels are used to send address and control information from the former node to the latter. The data channels are where data to be shared is positioned, and their direction depends on the action (read or write). A master reads information from a slave and writes data to it. Read response information is placed on the read data channel, while write response has a dedicated channel named write response channel. The master can verify that a write transaction has been completed through this action. A transaction is an exchange of data and includes the address and control information, data sent (request), and data received (response).

To increase the interface's bandwidth, the use of separate address and data channels for read and write transfers help (see Figure 2.10). There is no timing relationship between the groups of read and write channels. This means that a read sequence can happen simultaneously as the write one.

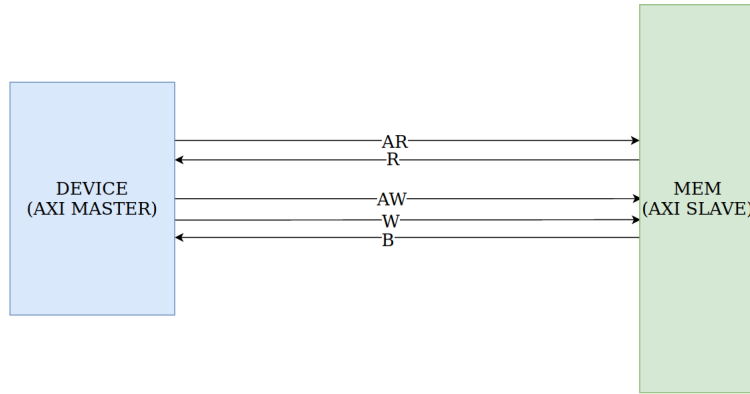


Figure 2.10: AXI channels

The following Tables provide AW and AR signals with their directions and a short description mentioned in the AXI Specification [4]. We focus on only these channels due to these are the channels that each Address Translation Unit performs the translation. Table 2.2 focuses on the AW, and Table 2.3 on the AR channel.

Write Address (AW)		
Name	Direction	Description
AWID	Master→Slave	Write address ID. This signal is the identification tag for the write address group
AWADDR	Master→Slave	Write address. The write address gives the address of the first transfer in a write burst transaction.
AWLEN	Master→Slave	Burst length. The burst length gives the exact number of transfers in a burst. This information determines the number of data transfers associated with the address.
AWSIZE	Master→Slave	Burst size. This signal indicates the size of each transfer in the burst.
AWBURST	Master→Slave	Burst type. The burst type and the size information determine how the address for each transfer within the burst is calculated.
AWLOCK	Master→Slave	Lock type. Provides additional information about the atomic characteristics of the transfer.
AWCACHE	Master→Slave	Memory type. This signal indicates how transactions are required to progress through a system.
AWPROT	Master→Slave	Protection type. This signal indicates the transaction's privilege and security level and whether it is data access or instruction access.
AWQOS	Master→Slave	Quality of Service, Quality of Service. The Quality of Service identifier is sent for each write transaction.
AWREGION	Master→Slave	Region identifier. Permits a single physical interface on a slave to be used for multiple logical interfaces.
AWUSER	Master→Slave	User signal. Optional User-defined signal in the write address channel.
AWVALID	Master→Slave	Write address valid. This signal indicates that the channel is signaling valid write address and control information.
AWREADY	Slave→Master	Write address ready. This signal indicates that the slave is ready to accept an address and associated control signals.

Table 2.2: Signals of AXI's Write Address (AW) channel

Read Address (AR)		
Name	Direction	Description
ARID	Master→Slave	Read address ID. This signal is the identification tag for the read address group of signals.
ARADDR	Master→Slave	Read address. The read address gives the address of the first transfer in a read burst transaction.
ARLEN	Master→Slave	Burst length. This signal indicates the exact number of transfers in a burst.
ARSIZE	Master→Slave	Burst size. This signal indicates the size of each transfer in the burst.
ARBURST	Master→Slave	Burst type. The burst type and the size information determine the calculation of each transfer's address within the burst.
ARLOCK	Master→Slave	Lock type. This signal provides additional information about the atomic characteristics of the transfer.
ARCACHE	Master→Slave	Memory type. This signal indicates how transactions are required to progress through a system.
ARPROT	Master→Slave	Protection type. This signal indicates the transaction's privilege and security level and whether it is data access or instruction access.
ARQOS	Master→Slave	Quality of Service, Quality of Service. Quality of Service identifier sent for each read transaction.
ARREGION	Master→Slave	Region identifier. Permits a single physical interface on a slave to be used for multiple logical interfaces.
ARUSER	Master→Slave	User signal. Optional User-defined signal in the read address channel.
ARVALID	Master→Slave	Read address valid. This signal indicates that the channel is signaling valid read address and control information.
ARREADY	Slave→Master	Read address ready. This signal indicates that the slave is ready to accept an address and associated control signals.

Table 2.3: Signals of AXI's Read Address (AR) channel

2.3.3 AXI Read Transactions

A master device, in order to read some data from the slave one, has to:

- Send the read address on the Read Address (AR) channel to inform the slave.

- In the good case scenario, the slave sends data from the requested address to the master through the Read Data (R) channel; contrariwise, it sends an error message on the same channel. An error occurs if, for instance, the address is invalid, the data is corrupted, or the access does not have the correct security permission

2.3.4 AXI Write Transactions

A master device, in order to write some data to the slave one, has to:

- Send an address on the Write Address (AW) channel and assign the data that wants to store on the slave on the Write Data (W) channel.
- Then, the slave sends back to the master the write response allowing it to know whether the transaction was successful or not.

2.3.5 Requirements

Below, one can find a number of the most critical requirements that the AXI Specification defines.

- When a VALID signal is asserted, it must remain asserted until the rising clock edge after the slave asserts the READY.
- The VALID signal of the AXI interface must not be dependent on the READY signal.
- A write response must always occur after the transaction's last write transfer.

2.4 Related Work

2.4.1 ARM IOMMU (System MMU)

System Memory Management Unit (SMMU) is the implementation of ARM's IOMMU as introduced by ARM Architecture Virtualization Extensions. SMMU performs address translation of an incoming AXI virtual address and AXI ID to an outgoing physical address. That is accomplished based on address mapping and memory attribute information held in translation tables. The SMMU architecture supports the concept of translation regimes, in which required memory access might require two stages of address translation. More information around the different SMMU's stages could be found in [5]. The central concept is that any memory accessed by a Guest OS or by an application requires two translation stages that together define a single translation regime. Stage 1 translates the VA to Intermediate Physical Address (IPA), and Stage 2 translates from IPA to PA.

An overview of an address translation process contains (a) the security state determination, (b) the address translation, (c) the memory access permissions and

determination of memory attributes, and (d) the memory attribute check. More precisely, the Security State Determination process identifies whether a transaction is from a Secure or Non-secure device. The Context Determination process identifies the Stage 1 or Stage 2 context resources that the SMMU uses to process a transaction. To find out the appropriate translation context, a Stream Identifier (StreamID) associates the transaction with a transaction stream. A Transaction Stream is a sequence of transactions associated with a particular thread of activity in the system and is associated with the same translation context. The StreamID uniquely identifies the originator of a transaction.

2.4.1.1 The Stream mapping table

In System MMU, the Stream mapping table maps each transaction to a transaction stream and its corresponding translation context. A System MMU implementation supports one of the following stream mapping schemes:

- StreamID matching: The StreamID is looked up in the set of Stream Match Registers (SMMU_SMR_n). When a unique match is found, the corresponding Stream-to-Context register (SMMU_S2CR_n) holds the stream's context.
- StreamID indexing: StreamID is a direct index to the required Stream-to-Context Register (SMMU_S2CR_n). If the StreamID is *m*, the required Stream-to-Context register is SMMU_S2CR_m.
- Finally, version 2 of SMMU defines a third scheme named Compressed StreamID Indexing that requires StreamID Compressed Indexing extension to be enabled: In this scheme, the StreamID is an indirect index to the required SMM_S2CR_n as follows: (a) The StreamID, *m*, indexes a single-byte S2CR Indexi field in the array of SMMU COMPINDEX_n registers. (b) The S2CR Indexi field holds the value of the SMMU S2CR_n for the stream. (So, if the value of the S2CRIndexi field is *m1*, the required Stream-to-Context register is SMMU_S2CR_{m1}). The Compressed StreamID indexing matching algorithm could be found in A.

2.4.1.2 Translation Context

A Translation Context provides information and resources required by the System MMU to process a transaction. The System MMU can process multiple transaction streams from different threads of execution and supports multiple live translation contexts. A translation context bank includes:

- State for configuring the translation process
- Capturing fault status and operations for maintaining cached translations

A Translation context bank specifies:

- The translation table base addresses
- Memory attributes to use during the translation table walk
- Translation table attribute remapping

The Translation context bank format depends on whether it is used for stage 1 or stage 2 translation. A translation context bank is arranged as a table in the SMMU configuration address map. Each entry in the table occupies a 4 KB or 64 KB address space. The System MMU architecture provides space for up to 128 Translation context banks. Each context bank of the SMMU can be considered as a one-page table – to be more accurate, each context bank has a field that points to a unique page table for this context. Each context bank has the SMMUCBnTTBR_m, where *m* can be 0 or 1. TTBR0, known as the Translation Table Base Register 0, holds the base address of translation table 0. For each context bank, the Translation Control Register, called SMMUCBnTCR, determines translation properties, including which one of the Translation Table Base Registers, SMMUCBnTTBR_m, defines the base address for the translation table walk required when an input address is not found in the TLB. An extension of the SMMUCBnTCR exists with the name SMMUCBnTCR2, which extends the SMMUCBnTCR by adding control information about the translation granule size and the size of the intermediate physical address.

Chapter 3

Design and Implementation

3.1 Overview

As mentioned in Chapter 1, the IOMMU is a hardware piece that serves almost the same purpose as a regular MMU. This purpose is to translate virtual addresses into physical ones. While the MMU is tightly coupled with the CPU (e.g., on Ariane RISC-V core, the MMU is located on the Execution pipeline stage), the IOMMU is located outside the core. Figures 3.1 and 3.2 presents designs that provide the potential location of the IOMMU inside a system. In general, IOMMUs allow peripheral devices, just like the CPUs does via standard MMUs, to benefit from the Virtual Memory. The Virtual Memory abstraction level's key benefits are mentioned in this thesis's abstract and introduction. If one wishes to focus on the benefits from an external device's perspective, the first one would be the security that IOMMU offers. The IOMMU ensures that an external device can not access any memory region that is not allowed for a specific device. Another key benefit is that many peripherals devices have fewer memory address bus bits than the rest of the system. The IOMMU fills this gap by translating narrow device addresses into the system's wider addresses. By doing that, I/O devices can address the entire memory through the IOMMU. For example, an x86 computer can address more than 4 gigabytes of memory with the Physical Address Extension (PAE) feature [9]. The way an IOMMU translates VAs to PAs is, in general, identical to the way an MMU performs the same task. More specifically, it searches on/through the Page Tables of a specific process to find the appropriate leaf PTE. The Page Tables are stored on the MM of the system, and copies are also available on the CPU's Data Cache. Lastly, when the appropriate leaf PTE is found, the translation action occurs if the permission checking allows it.

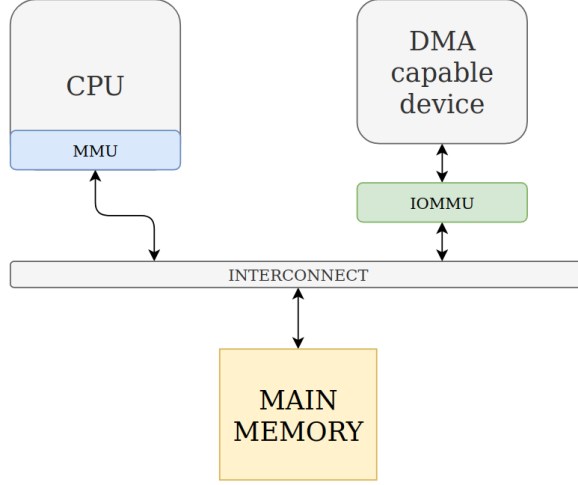


Figure 3.1: An example of where an IOMMU could be located in a system.

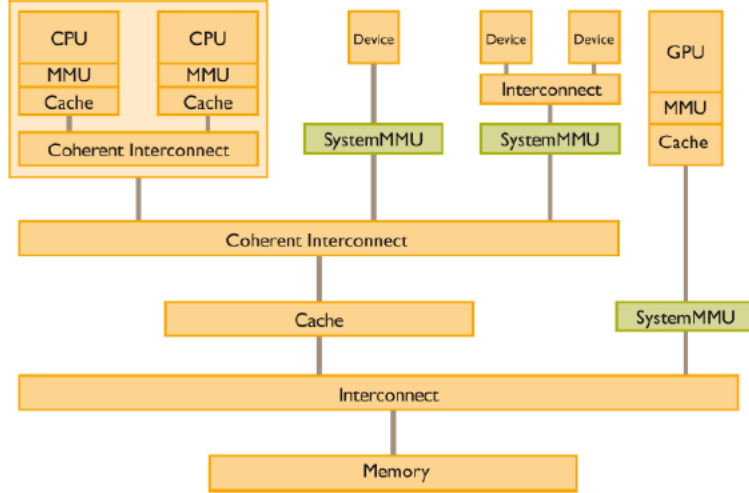


Figure 3.2: Examples of where an SMMU could be located in a system.

Source: ARM

3.2 Design Overview

In this thesis, we design and implement a generic and scalable IOMMU compatible with the RISC-V ISA, supporting its Sv39 and Sv48 modes as defined in the privileged manual [10]. We use SystemVerilog as the hardware description language [2]. On this implementation, the user can customize the number of input ports. On each input port, one peripheral could be connected. The communication protocol that the IOMMU supports is the AXI-4 AMBA [4, 3]. Nevertheless,

the implementation could be extended to support several different communication protocols, such as the AXI-3 AMBA. The IOMMU could support this protocol if the user grounds the two flag values (AR_ROB, AW_ROB) related to the reorder action for each one of the incoming channels. In general, these two flag values are responsible for enabling (disabling) the in-order (out-of-order) serving of the incoming requests for each one of the channels. This action is depending on the communication protocol requirements the user wants to use. To accomplish all requirements of the AXI-4 protocol, we have to enable the reordering logic. If we choose the previous version of AXI-4, the AXI-3, we will disable the reordering to maximize the IOMMU's performance.

Moreover, all sizes of the caches, the TLBs, the FIFOs, and others are parametric, allowing our implementation to adapt to any given system's requirements efficiently. Finally, another parametric feature of our implementation is that it allows the user to enable or not input buffers used for storing incoming requests. If enabled, one clock cycle is added to ATU's pipeline.

The central concept of our RV-IOMMU is that we separate the unit that accepts the incoming requests (only the channels that contain the address fields) and the unit that serves the missing requests by searching on the MM of the system using an AXI-Lite interface. The former unit is called Address Translation Unit (ATU), whereas the latter Address Translation Controller (ATC). These units communicate through two custom-made switches utilizing a custom-made protocol based on a handshake action (Valid / Ready). The Request Switch is responsible for connecting many ATUs with a single ATC and transfer their's missing requests to the ATC for serving. Based on the missing type, two different requests exist. The first asks for information about the Stream Matching process (i.e., the process that matches the external AXI ID with a process ID – ASID in terms of RISC-V) when an SMU miss occurs, while the second asks for Translation information (also known as PTE) when a TLB miss occurs.

On the other hand, the Response switch accepts ATC's responses and forwards them to the appropriate ATU. In case of receiving an invalidation message, the Response Switch forwards it to all the connected ATUs when they are ready to receive it. Figure 3.3 presents some of the system configurations that our RV-IOMMU could generate with a unique ATC unit. However, as previously mentioned, our implementation could create user-defined design configurations with multiple ATCs such as those presented in Figure 3.4.

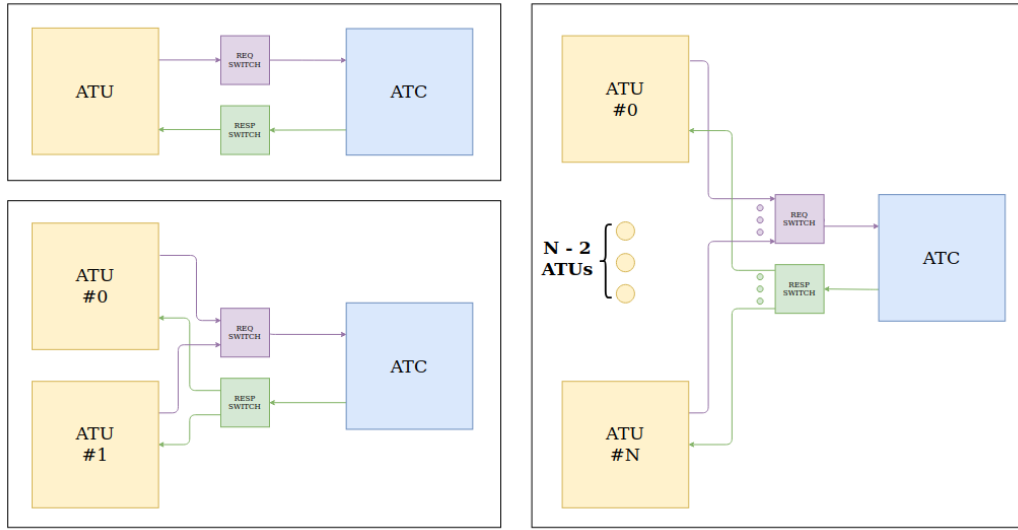


Figure 3.3: Abstract schematic of RV-IOMMU instantiation with a unique Address Translation Controller

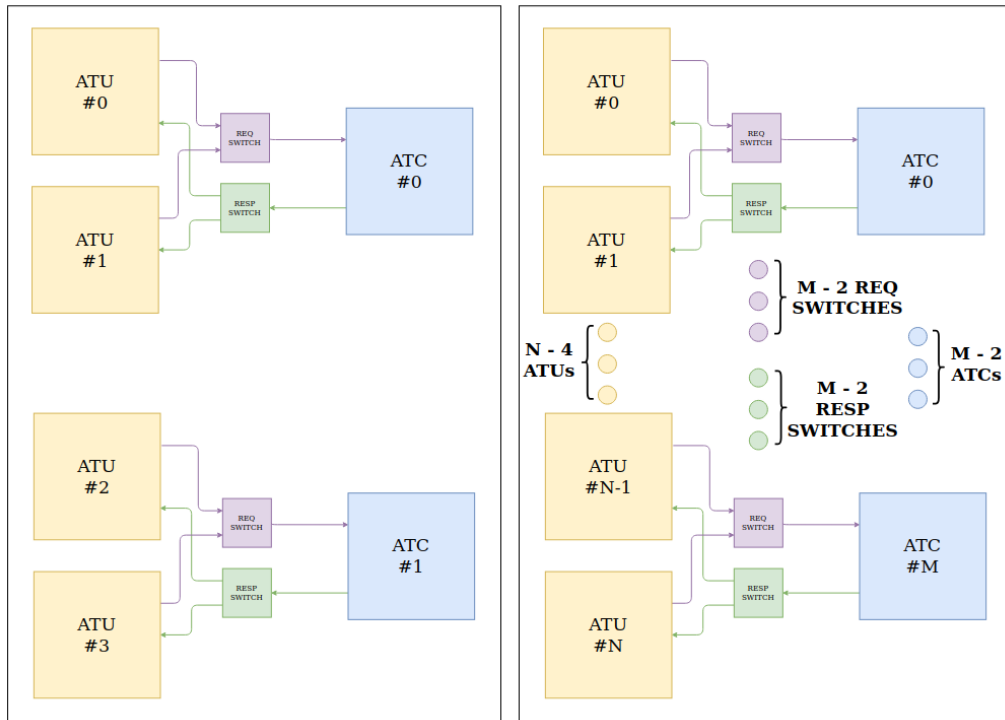


Figure 3.4: Abstract schematic of RV-IOMMU instantiation with many Address Translation Controllers

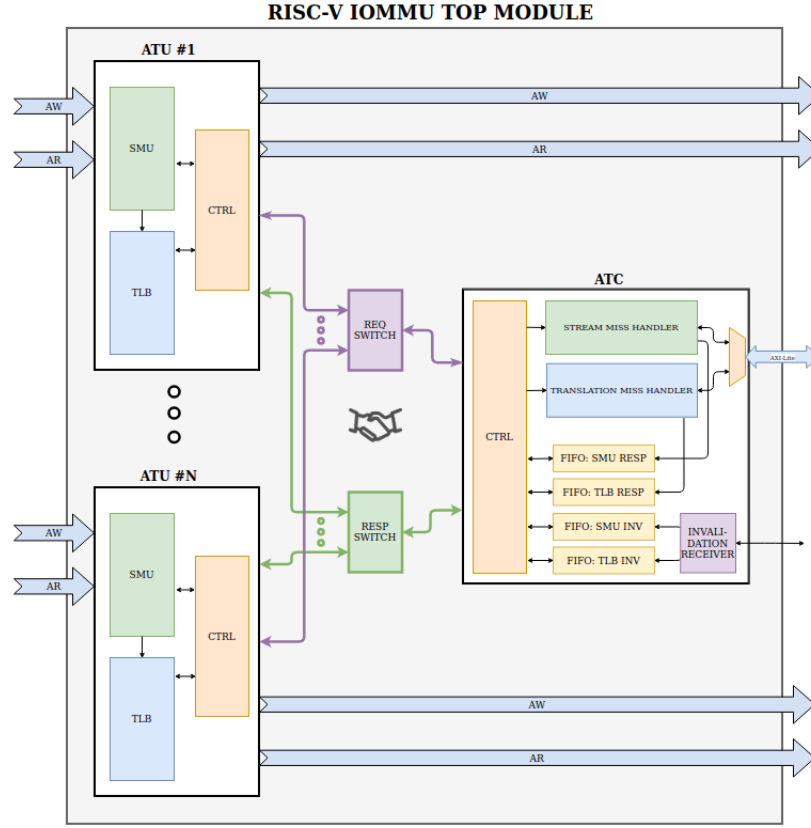


Figure 3.5: RV-IOMMU Overview

The RV-IOMMU is using two plus one main modules connected through switches, as depicted in Figure 3.5. These modules are:

- The Address Translation Unit (ATU): This module is responsible for accepting the incoming AXI AW & AR requests, performs the translation on the AWADDR & ARADDR busses, and outputs the translated requests. The RV-IOMMU supports one or more ATUs.
- The Address Translation Controller (ATC): This module is responsible for accepting the ATU(s) miss requests and serving them. To serve them, the ATC is connected with the MM using the AXI-Lite protocol.
- Internal switches: As above-mentioned, one or more ATUs can be connected with a unique ATC. To achieve this, we implement two kinds of switches, one that transfers the miss requests from ATUs to ATC and another for the responses.

3.3 Address Translation Unit

As shown in Figure 3.6, an ATU module is composed of three sub-modules, and its responsibility is to accept the incoming AXI requests and performs the translation. In particular, an ATU accepts only the AW and the AR channels of the AXI and performs the translation on the address fields. The rest of these two channels' data fields remain unchanged and “travel” through the ATU's pipeline accompanied by the addresses. As previously mentioned, it is easy to configure the number of the ATUs that will be generated in each implementation of RV- IOMMU by changing a specific parameter (NO_OF_ATU). That could generate as many input AXI channels, for translation, as the number of peripheral devices that one would like to connect on the RV-IOMMU. Before performing an address translation, it is crucial to report that the ATU performs permission checking to distinguish whether it is permitted or not. For example, it is not allowed to write on an address when the Write bit of the relevant PTE is not equal to one. In this case, the ATU asserts the permission error signal and does not translate it for this request (see 3.3.2.7). Instead, it returns an AXI Error message to the master of this request.

The ATU needs to complete two steps to perform a translation. The first step is to match the AXI ID of the incoming request with a specific process ID or, in terms of the RV, an ASID, and identify the PTBA. These, alongside the VA, are being used as an input to the second step of the process, i.e., the IO-TLB. IO's TLB responsibility is to accept the ASID, PTBA, and VA and, based on them, translate the VA to PA. If, however, the TLB does not currently store the relative PTE, it will miss. When a miss occurs, the TLB informs the ATU Controller (ATU CTRL). Following, the latter is responsible to (a) transfer this miss request to the ATC and (b) accept the response from the ATC and forward the appropriate data to the TLB (e.g., the PTE). The three main ATU's modules are:

- Stream Matching Unit (SMU)
- Translation Lookaside Buffer (TLB)
- ATU Controller CTRL

A detailed description of the aforementioned modules can be found in the following subsections. Figure 3.7 depicts ATU's Timing Diagram.

Moreover, ATU's Timing Diagram is depicted in Figure 3.7.

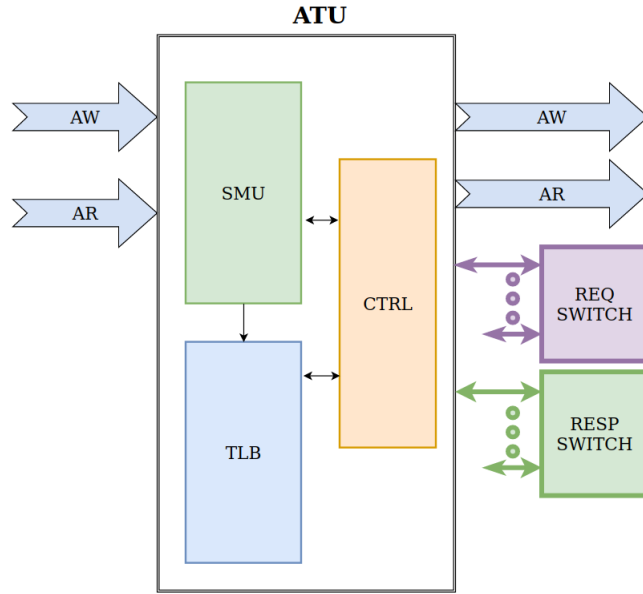


Figure 3.6: Address Translation Unit Overview

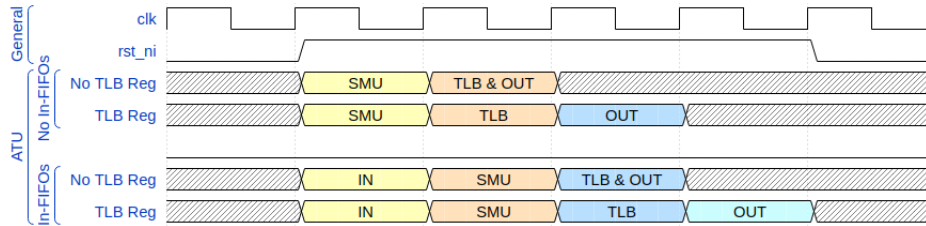


Figure 3.7: Address Translation Unit Timing Diagram

3.3.1 Stream Matching Unit

The first module that accepts the incoming AW and the AR channels of the AXI is the SMU. Its main task is to match the external AXI IDs from AW and the AR with the appropriate ASID followed by the PTBA. As shown in Figure 3.8, the main modules of the SMU are:

- (Optional) AR-FIFO & AW-FIFO
- Uniform AXI struct
- Input CTRL
- AXIID to ASID Set Associative Cache
- Replay (Duplicate) FIFO

- Unique FIFO
- Response Registers
- Registers that holds the send data on the ATC
- Output CTRL

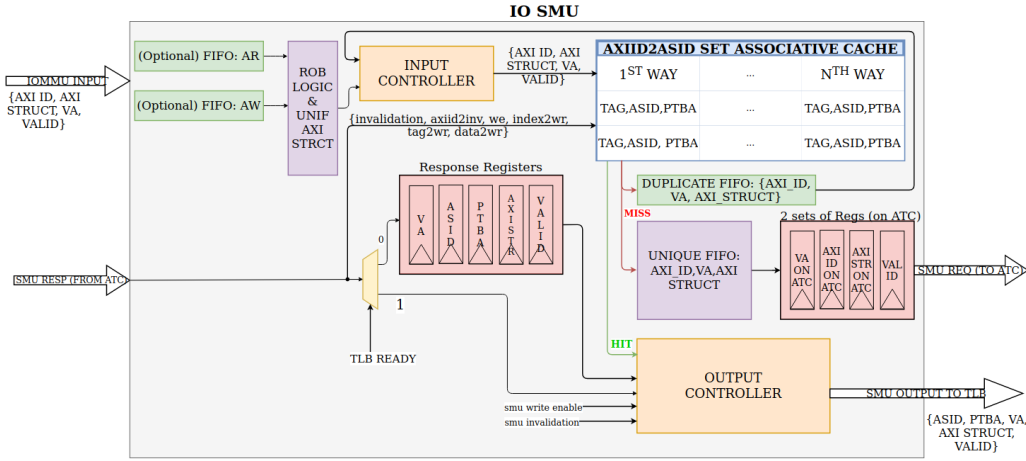


Figure 3.8: Stream Matching Unit

3.3.1.1 (Optional) AR-FIFO & AW-FIFO

These two synchronous FIFOs are generated if the user defines it and are responsible for storing the ATU's incoming AW and AR requests. The ATU is ready to accept new requests from the master device only when the corresponding FIFO is not full. The size of each FIFO is customizable, as users can define. Each FIFO is connected to the Uniform AXI Struct component, where it dequeues its data.

3.3.1.2 Uniform AXI struct

This module's task is to convert the dequeued requests to a new uniform struct. This uniform struct includes the common fields of the two previously mentioned channels (i.e., AW and AR): ID, address, etc. Moreover, it generates new fields different from those included in AW and AR. These fields are: (a) the `is_aw`, a 1-bit field that indicates if the request was read or write, and (b) the `ROB_ID`, which indicates in which position of the output FIFO the translated request will be stored. It is crucial to mention that the ROB ID field per channel is optional as it is utilized only if the Boolean parameter(s), `ROB_AW` and/or `ROB_AR`, is/are set to True. If the latter applies, our implementation reorders the requests to follow the flagged channel's initial order. Nevertheless, for our implementation not to violate the AXI-4 specification, reordering is mandatory for both channels.

3.3.1.3 Input CTRL

This module is located right before the AXIID to ASID Set Associative Cache, and its task is to choose the cache's input for each clock cycle. It can receive input from two different components, namely (a) the Uniform AXI struct (b) the Replay FIFO (described in Section [3.3.1.6]). The former is the most common choice. When a write/refill action occurs on the cache during a one clock cycle, an implemented mechanism is responsible for checking if the Replay FIFO's head entry exists in the unique FIFO. If true, the input controller continues to receive input from the Uniform AXI Struct. If not, the input controller switches its input method to the Replay FIFO one, if and only if the FIFO is not empty, at the next clock cycle. This will only stop if : (a) a miss at the cache occurs or (b) the Replay FIFO becomes empty. This logic is implemented through a Finite-State Machine (FSM).

3.3.1.4 AXIID to ASID Set Associative Cache

This module is a set-associative cache with a parametric number of ways and size. This cache will be a direct-mapped if the user configures its ways to be equal to one. For indexing on it, the implementation checks the user's parameters and uses a number of the MS bits of the AXI ID based on them. This index defines the line that an entry will be stored and where the implementation will search for it. The selection of the way that an entry will be stored is in a pseudo-random approach. This approach is basing on the Linear-Feedback Shift Register (LFSR) with a width equals to the number of configured ways. Another feature of our implementation of RV-IOMMU is the invalidation of the data stored on the local caches. For the invalidation of the Stream Matching entries, we implement a mechanism to discover the cache way that the entry for deletion is stored. An alternative would be to delete the entire cache set. However, this could lead to undesired results, such as the deletion of valid entries that could lead to potential cache misses. The mechanism mentioned above for invalidation spends two clock cycles to tackle this issue but eventually produces the desired result, i.e., to delete only the desired entry. To achieve that, it spends the first clock cycle finding the way the entry we want to delete is stored and, if found, it uses the second clock cycle to invalidate it by grounding the valid bit. We implement the set-associative cache to have two arrays of Static Random Access Memory SRAM cells for each way (see Figure 3.9). Below one can find details for the width of many cache-related entries, such as:

- The width of TAG = AXI ID WIDTH – AXI ID TO ASID Cache Index Width – AXI ID TO ASID Cache Offset Width.
- The width of AXI ID is predefined, and it depends on the system that we include the RV-IOMMU.

- The width of AXI ID TO ASID Cache Index = $\log_2(\text{Number of cache sets})$, which the number of cache sets is a parametric value.
- The width of ASID is a parameter defined by the RV core provided in the system. The maximum width is 16-bits for both SV39 and SV48 [10].
- Page Table Base Address (PTBA) width is 44-bits and is only the PA's effective bits. The width of PA is 56-bits.

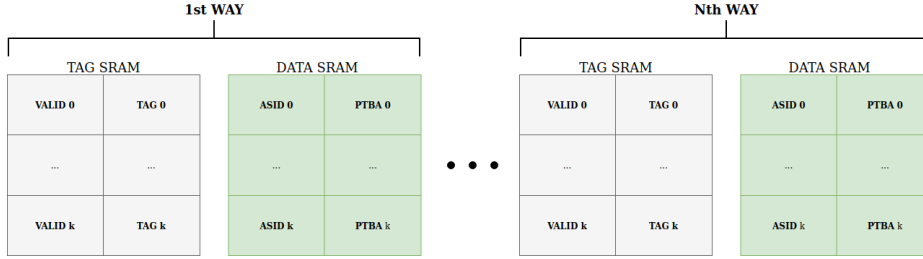


Figure 3.9: AXI ID to ASID Cache Architecture

3.3.1.5 Unique entries FIFO

The Unique Entries FIFO implementation is synchronous. The memory cells used in it are Flip-Flops and CAM cells, and its size is a user-defined parameter. It is responsible for storing the missed requests of the AXIID to ASID Set Associative Cache. This could only occur if the AXI IDs are not already stored in this FIFO or stored at the registers holding the data handled by the ATC. This FIFO store's structs consist of the AXI ID, the VA, and the uniform AXI Struct. The AXI IDs are stored on CAM cells while the rest are on Flip-Flops. The AXI IDs are stored on CAM cells to search all of them simultaneously when a potential enqueue occurs to decide if it will be accepted or not. The dequeued data of this FIFO are sending to the ATC through the Request Switch. More specifically, to avoid (a) transferring unwanted data (from ATC's perspective) and (b) to increase the width of wires from ATU to Request Switch only, the AXI ID is being transferred to the ATC whereas the remaining data – including the AXI ID – are being stored on the “Registers that hold the send data on the ATC” (see 3.3.1.7).

3.3.1.6 Replay (Duplicate) FIFO

It is a synchronous FIFO that stores the above's FIFO rejected data. Its size is user-defined, and the implementation uses Flip-Flops. The dequeued data of this FIFO are returned to the AXIID to ASID Set Associative Cache as shown in Figure 3.8 through the input controller. Lastly, the following attributes are being stored in this FIFO: (a) the AXI ID, (b) the VA, and (C) the Uniformed AXI struct.

3.3.1.7 Registers that hold data transmitted to the ATC

An ATU module contains two sets of these registers to improve its performance by sending a new request before storing the previous request's answer. These registers store the corresponding data of the AXI ID sent to ATC – including the AXI ID – to match the incoming responses with them. These data are the VA, the Uniformed AXI struct, and the Valid Register. When a response is received from the ATC, our implementation searches its corresponding data on these sets of registers and forwards them to the I/O TLB state of the pipeline. This occurs at the same clock cycle which the SMU received them.

3.3.1.8 Response Registers

These registers are being enabled when the IO-TLB pipeline stage is not ready to accept new requests. The data stored in the Response Registers are (a) the VA and Uniformed AXI struct (located in the registers described in Section 3.3.1.7) and (b) the ASID and PTBA from the incoming ATC response. When these registers store valid data, the ATU that includes them is not ready to accept new SMU responses from the ATC. It will become ready when the response registers become empty. The key idea is to avoid stalling the ATC serving misses process if an ATU's IO-TLB stage is not ready.

3.3.1.9 Output CTRL

This module chooses the SMU stage's output to feed the IO-TLB stage only when the former stage is ready to accept new requests. This selection is made according to a priority that following: (a) the hit answered data from AXIID to ASID Set Associative Cache, (b) only in case of receiving SMU invalidation this module checks if there are stored data on Response Registers to forward them to IO-TLB, (c) in case of receiving SMU missed data from the ATC forward them at the same clock cycle to IO-TLB as described on “Registers that holds the send data on the ATC” and (d) the data stored on Response Registers in case that are valid.

3.3.2 Input Output Translation Look-aside Buffer (IO-TLB)

The next stage of ATU's pipeline that accepts data from SMU is the Input-Output Translation Look-aside Buffer (IO-TLB) (see Figure 3.6). This unit's main task is to generate the physical address while performing the appropriate permission checking. The structure of this unit is similar to SMU's one. Figure 3.10 presents the main modules of the IO-TLB, which are:

- The Input CTRL
- The Translation Lookaside Buffer (TLB)
- A Unique FIFO

- A Replay (Duplicate) FIFO
- Registers that holds the send data on the ATC
- The Pipeline Register (Sv39/Sv48) (optional)
- The Construct Physical Address & Permission Checker (Sv39/Sv48)
- The re-construct AXI struct
- The AXI AW FIFO (ROB feature is optional)
- The AXI AR FIFO (ROB feature is optional)

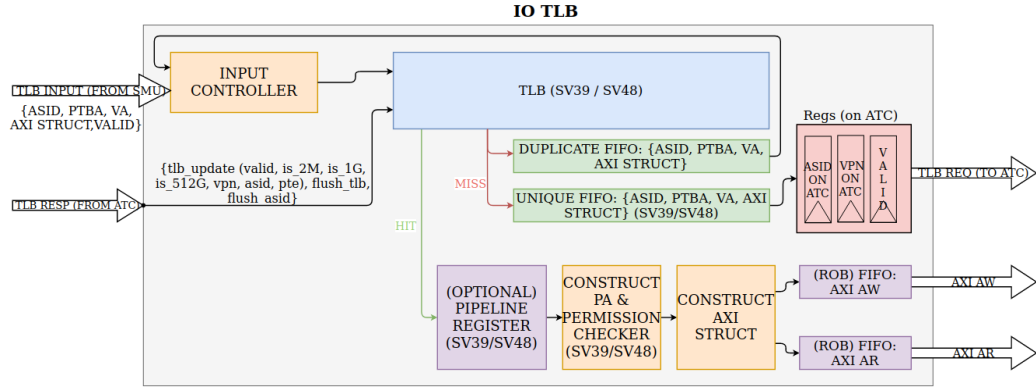


Figure 3.10: Input/Output Translation Lookaside Buffer

3.3.2.1 Input CTRL

The Input Controller is responsible for choosing the data that will forward to the TLB. This module can select between two different inputs. The first are data coming from the SMU. The second is the ones coming from the Replay FIFO 3.3.2.4. If SMU data are valid, the controller will redirect them to the TLB. If not, the redirected data will come from the Replay FIFO (supposing that this FIFO is not empty). Thus, this module can be seen as a simple multiplexer.

3.3.2.2 Translation Lookaside Buffer (Sv39/Sv48)

A TLB is a fully associative cache with a parametric size of entries that can receive two inputs. Inputs related to the translation process and input related to the flush and update process. The former is the ASID and the VA, whereas the latter is the flush_asid.i and update.i (which, in turn, includes a PTE, a VPN, an ASID, and some flags regarding the related page size). These inputs arrive at the TLB, which is the main task to find the appropriate PTE to construct the virtual address's translation, namely the physical address. Our TLB implementation is compatible

with both the Sv39 and the Sv48 specifications. The implementation identifies if invalid entries on TLB exist and chooses them in case of a replacement action. If all stored entries are valid, then we evict an entry using a Pseudo Least Recently Used Replacement policy.

In general, a TLB contains a CAM memory to hold the tags. Each tag consists of the ASID, the VPN3 (if Sv48 is enabled), the VPN2, the VPN1, VPN0, the is_512G (if Sv48 is enabled) flag value, the is_1G flag value, the is_2M flag value and the valid bit. Moreover, it stores PTE's data into separate memory. The implementation also supports invalidation and flushing actions. More specifically, there are two choices when it comes to flushing. The first is to flush the entire TLB and the second is to flush all entries related to a specific ASID.

3.3.2.3 Unique entries FIFO

The Unique Entries FIFO implementation follows a synchronous way. The memory cells used in it are Flip-Flops and CAM cells, and its size is a user-defined parameter. It is responsible for storing the missed requests of the TLB. That could only occur if the ASID concatenated with the effective bits of VA is not already stored in this FIFO or stored at the registers holding the data handled by the ATC. The structs that this FIFO stores consist of the ASID, the PTBA, the effective bits of VA (i.e., the VA without the 12-bit offset), and the write/read flag. The ASID and the VA's effective bits are stored on CAM cells while the rest of the fields on Flip-Flops. We use the CAM cells to store the ASID and the effective bits of VA to search all of them simultaneously when a potential enqueue action occurs to decide if this action will be accepted or not. The dequeued data of this FIFO are sending to the ATC through the ATU's controller first and the Request Switch after.

3.3.2.4 Replay (Duplicate) FIFO

This synchronous FIFO stores all entries that do not hit on the TLB even if they were also stored on TLB's Unique Entry FIFO. Its size is user-defined, and the implementation uses Flip-Flops. The dequeued data of this FIFO return to the TLB through the input controller (see Figure 3.10). Lastly, the Replay FIFO stores the following attributes: (a) ASID, (b) PTBA, (c) VA, and (C) Uniformed AXI struct.

3.3.2.5 Registers that hold data transmitted to the ATC

These registers store the ASID and the effective bits of VA. We store these to distinguish if the new-coming TLB misses will be, in turn, stored on the Unique Entries FIFO or not. When our implementation received the ATC response, it flushes the stored data on its Flips-Flops.

3.3.2.6 (Optional) Pipeline Register (Sv39/Sv48)

A pipeline register is a simple flip flop that exists only if the user enables the corresponding flag value (ATU_TLB_PIPELINE_REG). As shown in Figure 3.10, these registers store the TLB's output. The output consists of the following parameters: (a) the PTE, (b) flags about the page size, (c) the VA, and (d) the Uniform AXI struct. The parameters mentioned above will differ based on the enabled Sv.

3.3.2.7 Construct Physical Address & Permission Checker (Sv39/Sv48)

This combinatorial module aims to (a) check if the requested action is allowed and (b) to generate the PA. At first, this module decides, based on the uniform's AXI struct `is_wr` field, if the request's action is read or write. Based on it, it then checks if it is allowed to perform this action by reading the `pte.w` and `pte.r` field of a PTE (as mentioned in the RISC-V privileged specification [10]). The module generates the PA according to the previously mentioned specification if permitted. To support both the Sv39 and Sv48, we implement two versions of this module. If not permitted, the Construct Physical Address and Permission Checker module assigns a predefined out-of-bound address to the AXI struct. By doing this, the external AXI interconnect, or the AXI slave, returns an error response to the master. That allows us to assign the handling of the master's response to an external device. Even if it is neither elegant nor permanent, this solution was chosen due to time constraints. One of the future tasks would be implementing a more sophisticated mechanism to handle such situations (see Section 5).

3.3.2.8 Re-construct AXI struct

This module has only combinatorial logic, which re-generates the initial AXI request that the ATU receives. The only difference is that the `AWADDR` field (if it is a Write request) or `ARADDR` field (if it is a Read request) contains the translated PA instead of the VA. This module has two output ports. One is connected with the FIFO related with the Read requests and the other with the Write requests. Also, it outputs the ROB ID, which indicates at which position of the ROB FIFO the AXI struct will be stored (if the user has enabled the reordering feature).

3.3.2.9 (Optional ROB) AXI AW FIFO

As previously mentioned, our implementation allows the user to enable or not the reordering logic. If the ROB flag equals to one, this module is a Reorder Buffer that each new entry is inserted at a known position that `rob_id` indicates, and the dequeue is done as FIFO does it. In this way, we achieve the reordering that some communication protocols require, e.g., the AXI-4. If the external communication protocols do not have this requirement, this module becomes a typical synchronous FIFO that stores the translated requests (e.g., the AXI-3). The size is a parameter.

3.3.2.10 (Optional ROB) AXI AR FIFO

Same as above for the AR channel.

3.3.3 Controller Unit

This module communicates with the counterpart module of the ATC through switches. We separate this into two tasks. The first is to decide the request that will be sent to the ATC (for serving purposes). To accomplish this, it follows a static priority which gives priority first to the TLB misses and then to the full FIFOs. The second task is to receive the incoming responses from ATC's served requests. We achieve this by using input buffers (Flip-Flops) that store the response and forward them to the appropriate Unit (SMU or IO-TLB).

3.4 Address Translation Controller

As shown in Figure 3.11, an ATC is composed of three sub-modules, and its responsibility is to accept the missed requests from ATU(s) and then serve them. The ATC is connected with the MM of the system through an AXI-Lite interface to serve these requests. The missed requests could be sourced from the ATUs' SMU or TLB. When one missed request arrives at the ATC, the ATC recognizes the request's kind and forwards it to the appropriate handler. Such handler is either the Stream Miss Handler or the Translation Miss Handler. As we will see with more details in the following subsections, each of these handlers contains a cache (L2 SMU cache for the Stream Miss Handler and L2 TLB for the Translation Miss Handler). If a cache miss action occurs, the handler will search the MM entries. If a TLB entry could not be found again on the MM, the OS will generate it. The Stream Matching Table Finder performs the search action for a missing Stream Matching. For Translation misses, the Hardware Page Table Walker (HPTW) finds the appropriate PTE, which implements the algorithm described in Subsection 2.2.2. All responses are stored on the appropriate FIFO to transmit them to an ATU. Moreover, the ATC receives information about an invalidation action and forwards them to all connected ATUs through the Response switch. At the same time, it invalidates these entries on its local caches. The five main ATC's modules are:

- ATC Controller CTRL
- Stream Miss Handler (SMH)
- Translation Miss Handler (TMH)
- FIFOs

A detailed description of the modules mentioned above can be found in the following subsections. Figure 3.12 depicts ATC's Timing Diagram.

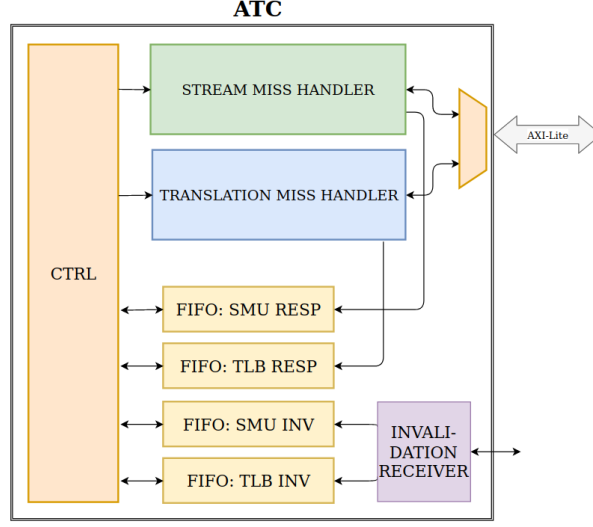


Figure 3.11: Address Translation Controller Overview

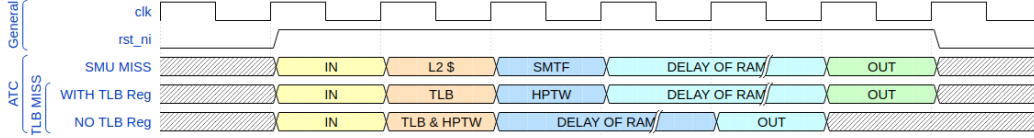


Figure 3.12: Address Translation Controller Timing Diagram

3.4.1 ATC Controller

This module is responsible for the communication between the ATU(s) and the ATC and contains logic about the entire unit's control. This controller's implementation is identical to the ATC's controller, with input buffers storing the incoming requests. It is also responsible for distinguishing a request to either SMU or TLB requests and forwarding them to the appropriate handler. Moreover, it is responsible for checking if an ATC's FIFO (SMU Response, TLB Response, SMU Invalidation, TLB Invalidation) contains valid data. If true, it forwards them to the Response Switch connected with this ATC. This controller enforces static priority. The highest of them belongs to the TLB Invalidation messages, then the SMU Invalidation messages, then the TLB Responses, and at last the SMU Responses. Our implementation provides the highest priority on the Invalidation messages to communicate the information around the implementation as soon as possible. By doing this, our implementation deletes all the stale data.

3.4.2 Stream Miss Handler

This module handles SMUs' missed requests, as previously mentioned. To achieve that, the Stream Miss Handler contains an L2 SMU set-associative cache and the Stream Matching Finder. The former contains the same parametric implementation with the 3.3.1.4, while the latter contains logic to calculate the address that stores the appropriate Stream Matching entry. The Finder is a very basic logic that follows a scalable policy. Its task is to add the AXI ID value with a predefined base address and search it on the MM. The communication between the Finder (AXI Master) and the MM (AXI Slave) is done through the AXI-Lite communication protocol.

3.4.3 Translation Miss Handler

This module is responsible for handling TLBs' missed requests. It contains a TLB and a Hardware Page Table Walker (HPTW). The former follows the same parametric implementation described in Section 3.3.2.2, and the latter handles TLBs' misses. The HPTW is a hardware component that follows the RISC-V privileged ISA [10] and searches the MM to identify the requested PTE. The HPTW implements in a hardware manner that algorithm presented in Section 2.2.2. That being said, the HPTW can communicate with the MM more than one time to identify the leaf PTE. The HPTW uses an AXI-Lite interface to communicate with memory and this permits the HPTW to be compatible with a plethora of memory blocks that have AXI interfaces. Moreover, the implementation supports both the 3-level page table walks required for SV39 and the 4-level page table walks required for SV48.

3.4.4 FIFOs

For the ATC implementation, our implementation generates four (4) synchronous FIFOs, where each one of them is customizing sized. These FIFOs are mentioned below in descending priority order:

- TLB Invalidation FIFO: Holds the TLB invalidation messages.
- SMU Invalidation FIFO: Stores the invalidation messages related with the SMU entries.
- TLB Response FIFO: Holds the TLB requests responses.
- SMU Response FIFO: Stores the SMU misses responses.

3.5 Communication between Address Translation Unit and Address Translation Controller

The communication between these two modules is being accomplished via switches. As depicted in Figure 3.5, two different kinds of switches exist. The Request switch (REQ SWITCH) is responsible for forwarding the ATU request(s) to the ATC while the Response Switch (RESP SWITCH) follows the other way around, i.e., receives the ATC responses and sends them back to the ATU. The communication protocol is custom-made and performs a handshake action (READY / VALID).

3.5.1 Request switch

This switch is connected with a parametric number of ATU(s) and forwards their request to a single ATC to serve them. Thus, this is an N-to-1 switch that transmits the data using combinatorial logic. The selection of the ATU's request forwarded to the ATC is being made through the Round-Robin starvation-free scheduling algorithm. More precisely, this switch contains a Flip-Flop that remembers the last ATU that accomplished a transaction. The Flip-Flop's purpose is to allow the Request switch to start searching from the next ATU of the one stored in it if this ATU is ready to make a transaction.

3.5.2 Response switch

This switch is connected with a single ATC and a parametric number of ATU(s). It receives the ATC's response message and forwards it only to the appropriate ATU (i.e., the ATU that generated the request). Thus, this is a 1-to-N switch that transmits its data through a combinatorial logic. The appropriate ATU selection is indicated by a field included in the response. It is crucial to mention here that if this switch's combinatorial logic identifies an invalidation message, it forwards it to all connected ATU(s) if they are in a ready state.

Chapter 4

Evaluation

The evaluation of our implementation includes two different phases. The first phase is the evaluation of our implementation in a RTL simulation environment whereas the second is performed on an actual hardware platform by placing our design on the Programmable Logic (PL) of an FPGA. During the second phase, we use Trenz, a hardware development board with a specific FPGA (part number: xczu9eg-ffvc900-2-e).

4.1 Phase 1: Simulation

During the first phase (i.e., the Simulation phase), we validate our implementation by simulating multiple components in an environment that includes a parametric instantiation of RV-IOMMU. To accomplish this, we implement the testbench environment of Figure 4.1. The total parametric approach of the implemented environment provides a user the ability to create its customized configuration. Some of the parameters related to the aforementioned testbench are: (a) the number of AW requests, (b) the number of AR requests, and (c) the read delay of Block RAM (BRAM). Following, the parameters related to the implementation of RV-IOMMU, also described in Chapter 3, are: (a) the number of ATU(s), (b) the TLBs sizes, (c) the L1 and L2 SMU cache size and associativity, (d) the FIFOs sizes and, optionally, (e) the pipeline register (on both of ATU and ATC). Finally, there are also parameters related to the user's needs. Such parameters, if enabled, allow the user to (a) enforce the in-order functionality for each one of the incoming channels and (b) if the incoming buffers of the ATU will be generated or not. Lastly, our testbench generates the requests and then forwards them to the RV-IOMMU. It also stores and validates the correctness of the responses. We achieve these two actions by implementing two (2) SystemVerilog tasks. More specifically, the response validity checking actions are achieved because the user knows a priori the Page Tables.

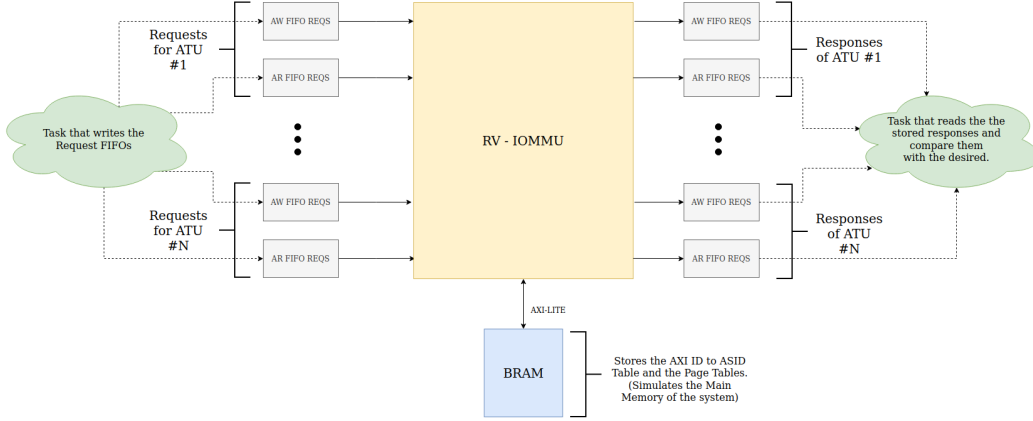


Figure 4.1: Abstract schematic of the simulation environment

Moreover, we utilize the environment mentioned above to measure the RV-IOMMU's performance. We simulate the design with a variety of configurations to obtain various measurements. These measurements include the clock cycles that each request spends to be served and some architecture metrics (e.g., the number of hits and misses on each cache and TLB). We add extra logic to the HPTW to understand the number of times the HPTW needs to be activated per run and – in each run – how many leaf PTEs of 4KB, 2MB, and 1Gb size were found on the RAM. The testbench's RAM simulates the MM of the System that, among others, contains the AXI ID to ASID Table and the Page Tables. More precisely, the former contains ten entries - supporting incoming AXI IDs for 0 to 9, whereas the latter contains one three-level, two two-level, and four one-level Page Tables. The rest of this section provides the results of some runs performed in this work.

4.1.1 Metrics

This subsection presents two measurement categories contributing to the overall evaluation of the implementation presented herein. The first is the required clock cycles needed to accomplish the run, and the second is architectural metrics. The measurements were performed for several configurations to explore the implementation's behavior and capabilities. The below-presented tables depict the parameters and the results of the measures. As one can observe, these tables include, among others, the implementation's TLBs and cache's misses and hits. More specifically, the calculation of the total clock cycles required for the translation of a request is measured from when the RV-IOMMU accepts this request until it outputs the translated response (we only calculate the cycles that the request remained in the RV IOMMU pipeline). It is crucial to mention that all the requests fed to the RV-IOMMU are pre-initialized inside a FIFO. Thus, there is always a new request to “feed” our RV-IOMMU. This measurement, even if it does not deliver our implementation's absolute performance, it provides an estimation of it.

Another factor taken into consideration to produce the measurements mentioned above is the requests' traffic pattern. The worst-case scenario could occur when all, or the majority of, the received requests' AXI ID and PTE does not exist on the corresponding caches/TLBs. As mentioned in Section 3, the AXI ID to ASID cache follows a pseudo-random replacement policy, whereas the TLB an LRU one. Considering this, we defined the worst-case scenario as the scenario that generates random requests using the \$urandom_range() function of SystemVerilog. More precisely, this function will decide both the AXI ID and the VA.

Conversely, the normal-case scenario occurs when all incoming requests are refer to a specific AXI ID, and their virtual addresses are sequential. We defined the normal-case scenario for our implementation to generate the AXI IDs randomly (by using the \$urandom_range() function of SystemVerilog) and the VAs to be in a sequential pattern. The first VA that the normal-case scenario generates is randomly chosen. Then, for every future request, the sequential pattern will add to the previously generated VA the decimal number 64 in order to model sequential cache-line accesses.

4.1.1.1 The performance cost of reordering

Herein, we estimate the cost of the reordering feature that our implementation provides on both the randomized and sequential traffic patterns. To calculate this, we choose not to enable ATU(s)' input buffers as we wanted to measure the exact time that each incoming request spends to serve.

Table 4.1 depicts the RV-IOMMU configurations tested under the randomized traffic generator. The required clock cycle measurements could then be found in Table 4.2 with references to their corresponding histograms. Finally, the architectural metric could be found in Table 4.3.

RV-IOMMU configuration (randomized traffic generator)						
	Conf. 1	Conf. 2	Conf. 3	Conf. 4	Conf. 5	Conf. 6
Number of ATU(s)	1	1	2	2	8	8
Total number of AW Reqs	2048	2048	4096	4096	16384	16384
Total number of AR Reqs	2048	2048	4096	4096	16384	16384
AR Reordering	✓	-	✓	-	✓	-
AW Reordering	✓	-	✓	-	✓	-
ATU Input Buffers	-	-	-	-	-	-
ATU TLB	✓	✓	✓	✓	✓	✓
Pipeline Reg						
ATC TLB	✓	✓	✓	✓	✓	✓
Pipeline Reg						
ATU L1 TLB Size	16	16	16	16	16	16
ATC L2 TLB Size	32	32	32	32	32	32
ATU L1 SMU	8	8	8	8	8	8
Cache Associativity						
ATU L1 SMU	2	2	2	2	2	2
Cache Sets						
ATC L2 SMU	16	16	16	16	16	16
Cache Associativity	Ways	Ways	Ways	Ways	Ways	Ways
ATC L2 SMU	16	16	16	16	16	16
Cache Sets						
RAM Read Latency	2cc	2cc	2cc	2cc	2cc	2cc

Table 4.1: RV-IOMMU configurations used to explore the cost of the reordering feature feed by a randomized traffic generator

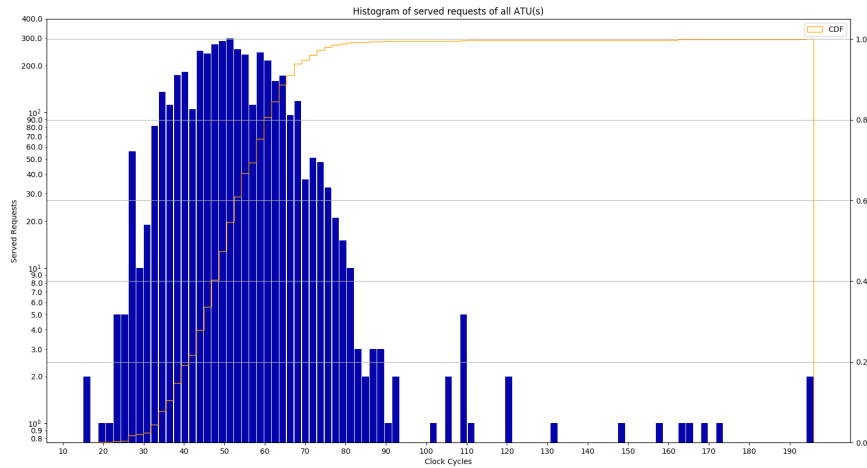
Results about the required clock cycles of Table 4.1						
	Conf. 1	Conf. 2	Conf. 3	Conf. 4	Conf. 5	Conf. 6
Average (50 th percentile)	51 cc	51 cc	102 cc	102 cc	415 cc	414 cc
95 th percentile	71 cc	70 cc	137 cc	136 cc	526 cc	524 cc
99 th percentile	81 cc	80 cc	207 cc	151 cc	560 cc	552 cc
Minimum value	15 cc	3 cc	29 cc	3 cc	60 cc	3 cc
Maximum value	196 cc	193 cc	580 cc	2046 cc	2392 cc	8913 cc
Histogram Figure	See 4.2	See 4.3	See 4.4	See 4.5	See 4.6	See 4.7

Table 4.2: Time (in cc) results of configurations of Table 4.1 used to explore the cost of the reordering feature (randomized traffic generator)

Results about the architectural metrics of Table 4.1						
	Conf. 1	Conf. 2	Conf. 3	Conf. 4	Conf. 5	Conf. 6
One ATU's L1 SMU cache hits	4096	4096	4069	4069	4069	4069
One ATU's L1 SMU cache misses	18	19	26	25	22	26
One ATU's L1 TLB cache hits	4096	4096	4096	4069	4069	4069
One ATU's L1 TLB cache misses	38453	38465	79875	79999	333347	333680
ATC's L2 SMU cache hits	0	0	8	7	44	45
ATC's L2 SMU cache misses	6	6	6	6	6	6
ATC's L2 TLB cache hits	53	64	110	128	526	523
ATC's L2 TLB cache misses	4027	4020	8007	7999	31965	31971
HPTW total searches on MM	4027	4020	8007	7999	31965	31971
# of 1G PG found by HPTW	1804	1784	3665	3645	14333	14380
# of 2M PG found by HPTW	902	921	1777	1762	7153	7010
# of 4K PG found by HPTW	1321	1315	2565	2592	10479	10581

Table 4.3: Architectural metrics of configurations of Table 4.1 used to explore the cost of the reordering feature (randomized traffic generator)

Disclaimer: Table 4.3 results, as well as measurements presented below, present a vast number of ATU's L1 TLB misses. This measurement is growing accord to the duration of the run. This occurs because the ATU's TLB's input controller (see 3.3.2.1) will always select to forward the Replay FIFO entries to the TLB, if invalid data are coming from the SMU. This is, of course, not optimal and can be enhanced in future work.



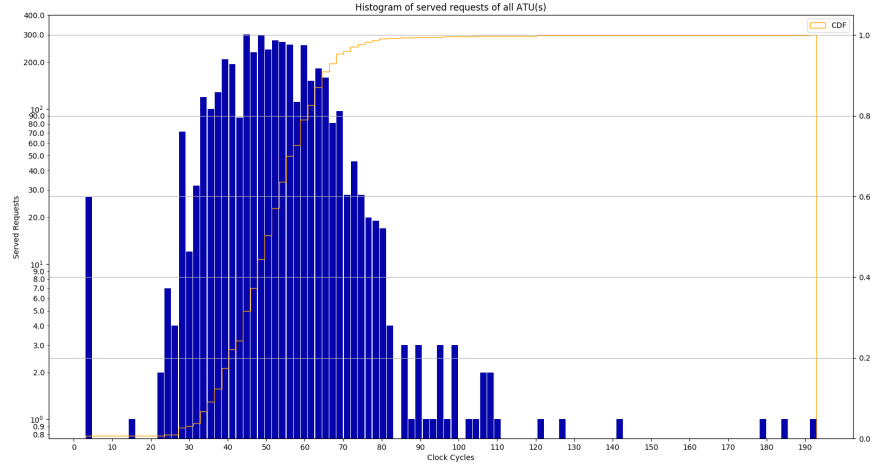


Figure 4.3: Histogram of clock cycles that RV-IOMMU requires to accomplish the run on Conf. 2 of Table 4.1

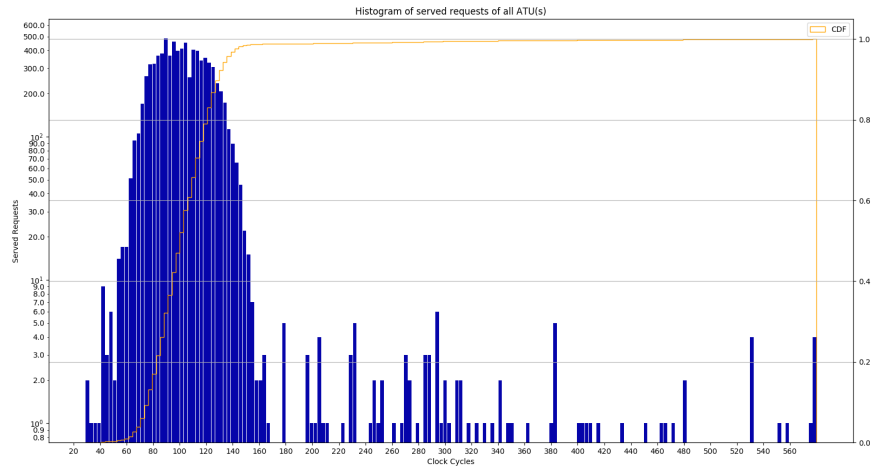


Figure 4.4: Histogram of clock cycles that RV-IOMMU requires to accomplish the run on Conf. 3 of Table 4.1

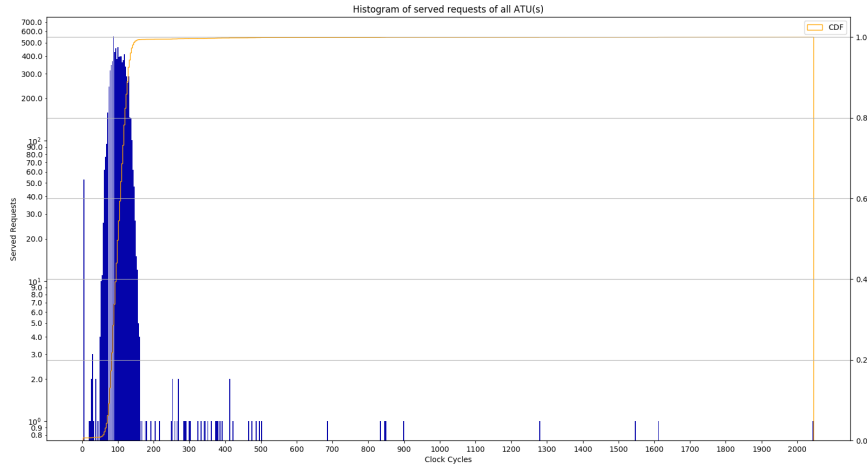


Figure 4.5: Histogram of clock cycles that RV-IOMMU requires to accomplish the run on Conf. 4 of Table 4.1

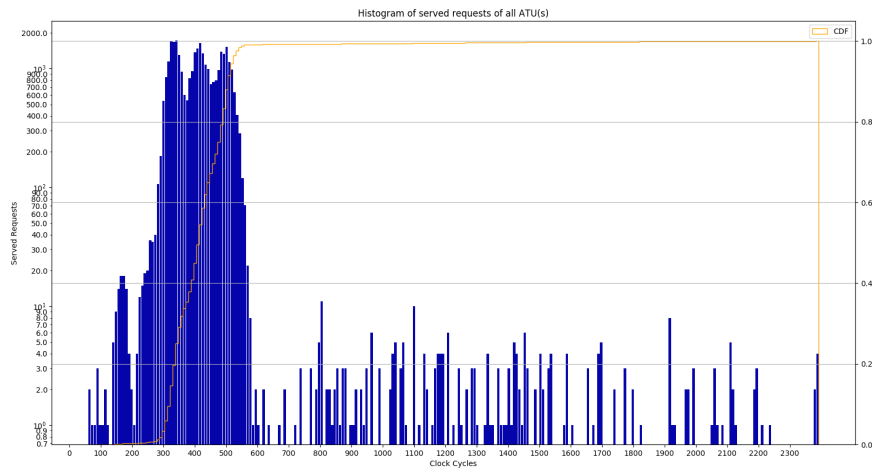


Figure 4.6: Histogram of clock cycles that RV-IOMMU requires to accomplish the run on Conf. 5 of Table 4.1

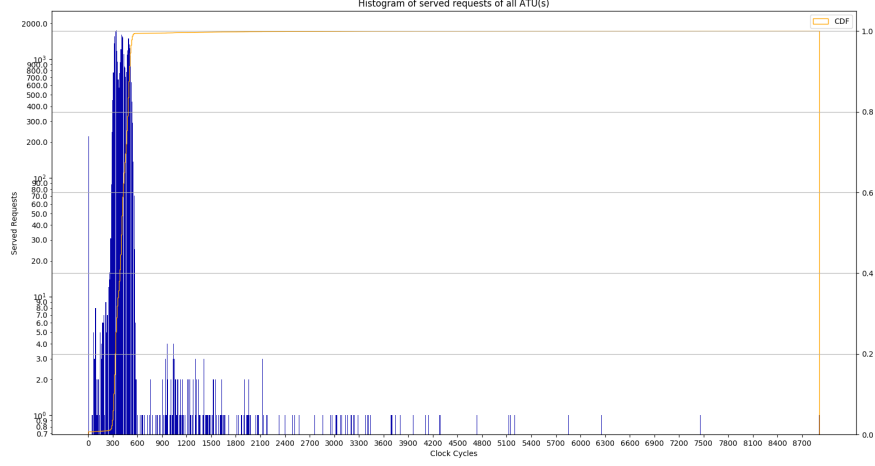


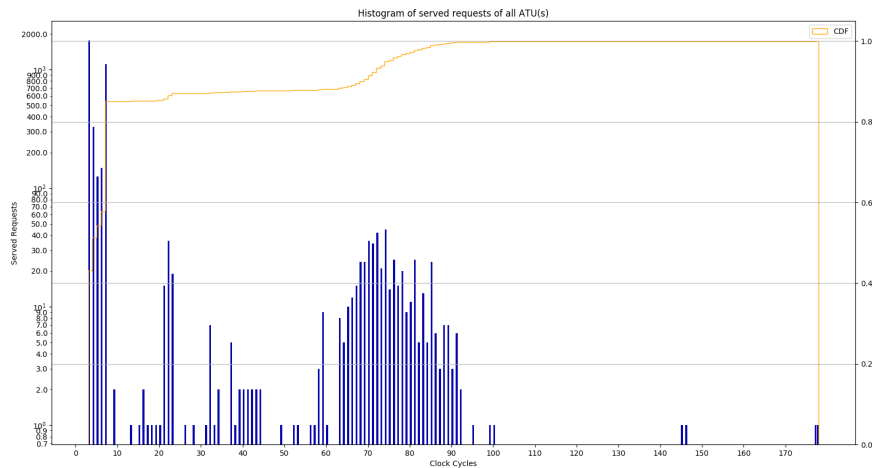
Figure 4.7: Histogram of clock cycles that RV-IOMMU requires to accomplish the run on Conf. 6 of Table 4.1

Table 4.2 and 4.3 compare the results between Configuration 1 and 2, Configuration 3 and 4, and Configuration 5 and 6. Concluding, based on the comparisons mentioned above, one could easily understand that there is no extra overhead when the Reordering feature is enabled if the incoming traffic follows a random pattern. This is because the cost of the large number of TLB misses is the main performance bottleneck for these configurations, resulting in the elimination of the reordering cost.

Table 4.4 depicts the tested RV-IOMMU configurations under the sequential traffic generator. Table 4.5 presents the required clock cycle measurements with references to their corresponding histograms. Finally, Table 4.6 shows the architectural metrics.

Configurations RV-IOMMU (sequential traffic generator)						
	Conf. 1	Conf. 2	Conf. 3	Conf. 4	Conf. 5	Conf. 6
Number of ATU(s)	1	1	2	2	8	8
Total number of AW Reqs	2048	2048	4096	4096	16384	16384
Total number of AR Reqs	2048	2048	4096	4096	16384	16384
AR Reordering	✓	-	✓	-	✓	-
AW Reordering	✓	-	✓	-	✓	-
ATU Input Buffers	-	-	-	-	-	-
ATU TLB	✓	✓	✓	✓	✓	✓
Pipeline Reg						
ATC TLB	✓	✓	✓	✓	✓	✓
Pipeline Reg						
ATU L1 TLB Size	16	16	16	16	16	16
ATC L2 TLB Size	32	32	32	32	32	32
ATU L1 SMU	8	8	8	8	8	8
Cache Associativity						
ATU L1 SMU	2	2	2	2	2	2
Cache Sets						
ATC L2 SMU	16	16	16	16	16	16
Cache Associativity	Ways	Ways	Ways	Ways	Ways	Ways
ATC L2 SMU	16	16	16	16	16	16
Cache Sets						
RAM Read Latency	2cc	2cc	2cc	2cc	2cc	2cc

Table 4.4: RV-IOMMU configurations used to explore the cost of the reordering feature feed by a sequential traffic generator



Results about the required clock cycles of Table 4.4						
	Conf. 1	Conf. 2	Conf. 3	Conf. 4	Conf. 5	Conf. 6
Average (50 th percentile)	4 cc	3 cc	4 cc	3 cc	7 cc	3 cc
95 th percentile	75 cc	9 cc	75 cc	15 cc	128 cc	59 cc
99 th percentile	85 cc	116 cc	89 cc	94 cc	205 cc	167 cc
Minimum value	3 cc	3 cc	3 cc	3 cc	3 cc	3 cc
Maximum value	178 cc	153 cc	203 cc	167 cc	826 cc	752 cc
Histogram Figure	See 4.8	See 4.9	See 4.10	See 4.11	See 4.12	See 4.13

Table 4.5: Time (in cc) results of configurations found in Table 4.4, used to explore the cost of the reordering feature (sequential traffic generator)

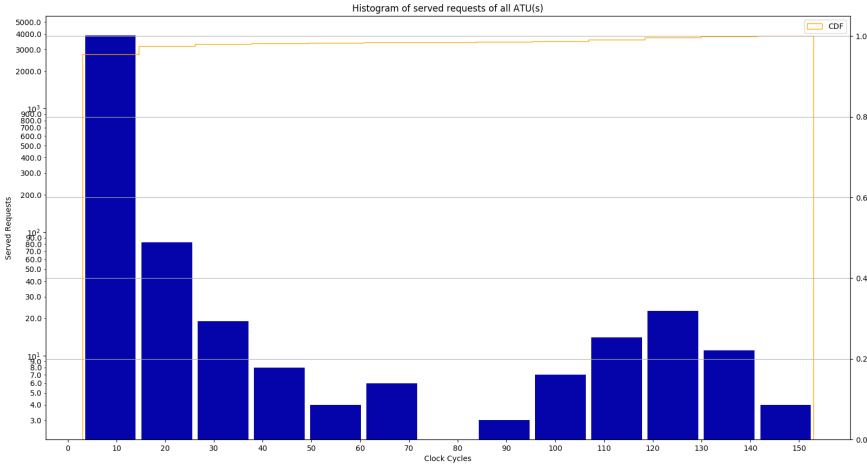


Figure 4.9: Histogram of clock cycles that RV-IOMMU requires to accomplish the run on Conf. 2 of Table 4.4

Results about the architectural metrics of Table 4.4							
	Conf. 1	Conf. 2	Conf. 3	Conf. 4	Conf. 5	Conf. 6	
One ATU's L1 SMU cache hits	4096	4096	4069	4069	4069	4069	
One ATU's L1 SMU cache misses	18	20	21	27	22	26	
One ATU's L1 TLB cache hits	4096	4096	4096	4069	4069	4069	
One ATU's L1 TLB cache misses	584	452	639	690	4135	4616	
ATC's L2 SMU cache hits	0	0	8	7	44	45	
ATC's L2 SMU cache misses	6	6	6	6	6	6	
ATC's L2 TLB cache hits	3	5	11	30	166	119	
ATC's L2 TLB cache misses	72	72	142	144	574	583	
HPTW total searches on MM	72	72	142	144	574	583	
# of 1G PG found by HPTW	4	4	4	10	13	31	
# of 2M PG found by HPTW	2	2	7	4	30	24	
# of 4K PG found by HPTW	66	66	131	130	531	528	

Table 4.6: Architectural metrics of configurations of Table 4.1 used to explore the cost of the reordering feature (sequential traffic generator)

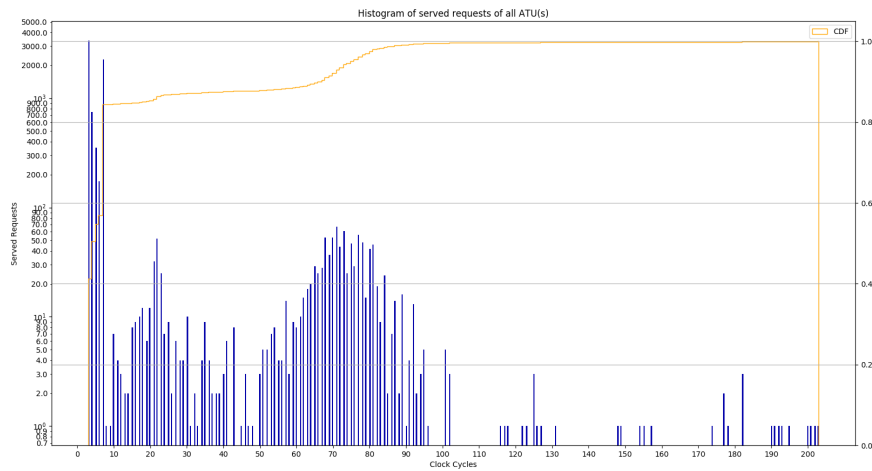


Figure 4.10: Histogram of clock cycles that RV-IOMMU requires to accomplish the run on Conf. 3 of Table 4.4

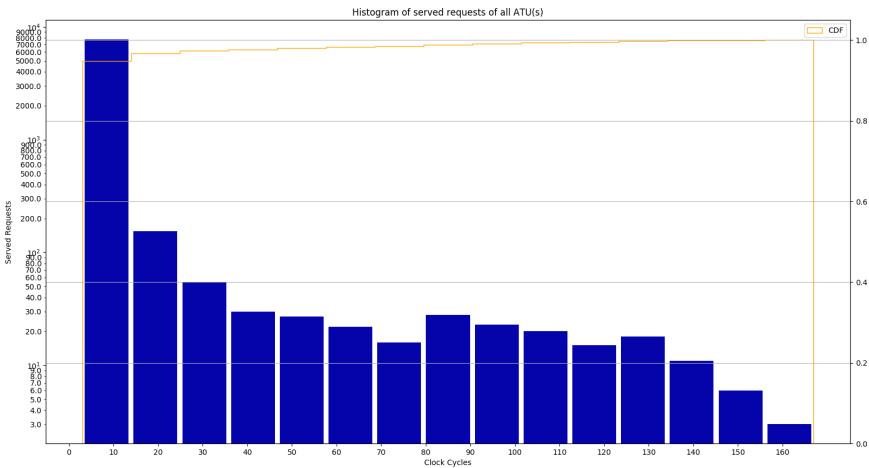


Figure 4.11: Histogram of clock cycles that RV-IOMMU requires to accomplish the run on Conf. 4 of Table 4.4

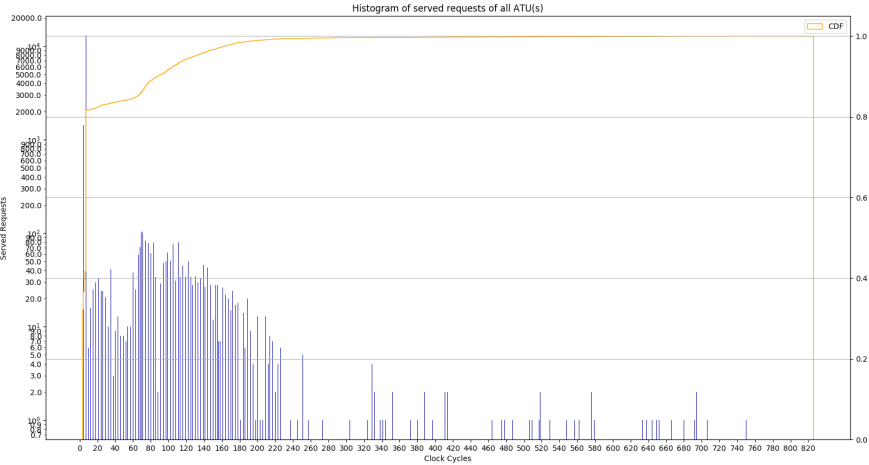


Figure 4.12: Histogram of clock cycles that RV-IOMMU requires to accomplish the run on Conf. 5 of Table 4.4

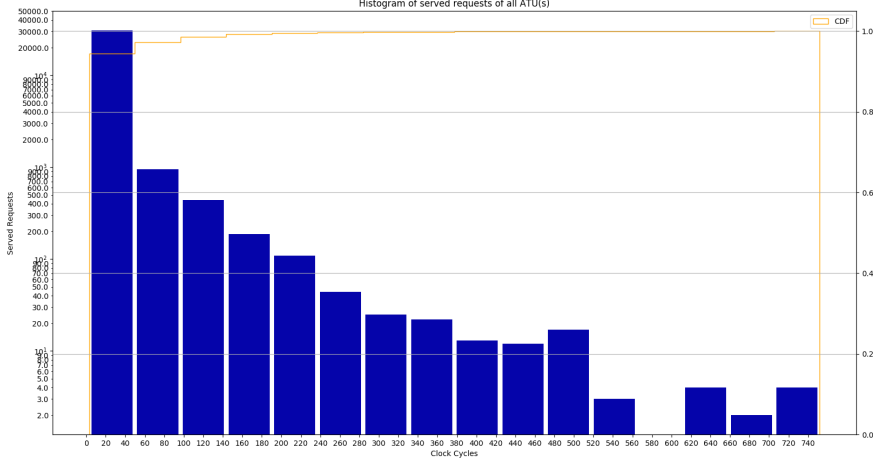


Figure 4.13: Histogram of clock cycles that RV-IOMMU requires to accomplish the run on Conf. 6 of Table 4.4

Table 4.5 and 4.6 compare the results between Configuration 1 and 2, Configuration 3 and 4, and Configuration 5 and 6. This comparison shows that the performance decreases when the reordering feature is enabled (if and only if the incoming traffic follows a sequential pattern). More precisely, if the reordering feature is enabled, the response will be stored at a specific position of the output FIFO (as described in 3.3.2.9) and will wait until its turn to be dequeued. This will cause a performance overhead as all potential hit responses will have to wait for a former miss to be served (all hits under miss requests). On the contrary, disabling the reordering feature would allow our implementation to output the responses immediately after their translation without stalling them, resulting in an overall performance gain.

4.1.1.2 Performance sensitivity analysis on ATU's TLB size

To analyse the performance sensitivity on different parameters of the RV-IOMMU, we change one RV-IOMMU parameter at a time. The remaining (i.e., ones that will not change) of the parameters will have realistic values. The first parameter that we will explore is ATU's TLB size by using the random pattern traffic generator. Since we explore the ATU's TLB size, we choose to have an RV-IOMMU with one ATU.

RV-IOMMU configurations (randomized traffic generator)				
	Conf. 1	Conf. 2	Conf. 3	Conf. 4
Number of ATU(s)	1	1	1	1
Total number of AW Reqs	2048	2048	2048	2048
Total number of AR Reqs	2048	2048	2048	2048
AR Reordering	✓	✓	✓	✓
AW Reordering	✓	✓	✓	✓
ATU Input Buffers	✓	✓	✓	✓
ATU Input Buffers Size	2 entries	2 entries	2 entries	2 entries
ATU TLB Pipeline Reg	✓	✓	✓	✓
ATC TLB Pipeline Reg	✓	✓	✓	✓
ATU L1 TLB Size	2	8	32	128
ATC L2 TLB Size	8	8	8	8
ATU L1 SMU Cache Associativity	8 Ways	8 Ways	8 Ways	8 Ways
ATU L1 SMU Cache Sets	4	4	4	4
ATC L2 SMU Cache Associativity	8 Ways	8 Ways	8 Ways	8 Ways
ATC L2 SMU Cache Sets	8	8	8	8
RAM Read Latency	100cc	100cc	100cc	100cc

Table 4.7: RV-IOMMU configurations used to explore the performance effects of ATU's L1 TLB size (random traffic generator)

Results about the required clock cycles of Table 4.7				
	Conf. 1	Conf. 2	Conf. 3	Conf. 4
Average (50 th percentile)	1550 cc	1550 cc	1550 cc	1450 cc
95 th percentile	1962 cc	1662 cc	1962 cc	1958 cc
99 th percentile	2166 cc	2162 cc	2162 cc	2064 cc
Minimum value	423 cc	423 cc	423 cc	417 cc
Maximum value	3614 cc	3614 cc	3614 cc	3614 cc

Table 4.8: Time (in cc) results of configurations of Table 4.7, used to explore the performance effects of ATU's L1 TLB size (random traffic generator)

Results about the architectural metrics of Table 4.7				
	Conf. 1	Conf. 2	Conf. 3	Conf. 4
One ATU's L1 SMU cache hits	4096	4096	4069	4069
One ATU's L1 SMU cache misses	9	9	9	9
One ATU's L1 TLB cache hits	4096	4096	4096	4069
One ATU's L1 TLB cache misses	793698	790608	781011	744642
ATC's L2 SMU cache hits	0	0	0	0
ATC's L2 SMU cache misses	6	6	6	6
ATC's L2 TLB cache hits	17	18	24	21
ATC's L2 TLB cache misses	4078	4067	4024	3872
HPTW total searches on MM	4078	4067	4024	3872
# of 1G PG found by HPTW	1811	1811	1795	1763
# of 2M PG found by HPTW	915	912	908	868
# of 4K PG found by HPTW	1352	1344	1321	1241

Table 4.9: Architectural measurements of configurations of Table 4.7 exploring the performance effects of the ATU's L1 TLB size (random traffic generator)

By observing tables 4.8, 4.9, and 4.7, one can identify no significant performance gain when one increases the ATU's L1 TLB size when the traffic pattern of incoming requests is random. However, this does not come as a surprise as the likelihood of a TLB miss for every new incoming request (generated from the random algorithm) is extremely high. Because of this behavior, every missed request will be served in the ATC's HPTW. Similar behavior is also occurring on the L2 TLB of ATC.

Tables 4.10, 4.11, and 4.12 explore the performance fluctuations when the TLB size changes and the sequential generator is utilized.

RV-IOMMU configurations (sequential traffic generator)				
	Conf. 1	Conf. 2	Conf. 3	Conf. 4
Number of ATU(s)	1	1	1	1
Total number of AW Reqs	2048	2048	2048	2048
Total number of AR Reqs	2048	2048	2048	2048
AR Reordering	✓	✓	✓	✓
AW Reordering	✓	✓	✓	✓
ATU Input Buffers	✓	✓	✓	✓
ATU Input Buffers Size	2 entries	2 entries	2 entries	2 entries
ATU TLB Pipeline Reg	✓	✓	✓	✓
ATC TLB Pipeline Reg	✓	✓	✓	✓
ATU L1 TLB Size	2	8	32	128
ATC L2 TLB Size	8	8	8	8
ATU L1 SMU Cache Associativity	8 Ways	8 Ways	8 Ways	8 Ways
ATU L1 SMU Cache Sets	4	4	4	4
ATC L2 SMU Cache Associativity	8 Ways	8 Ways	8 Ways	8 Ways
ATC L2 SMU Cache Sets	8	8	8	8
RAM Read Latency	100cc	100cc	100cc	100cc

Table 4.10: RV-IOMMU configurations used to explore the performance effects of ATU's L1 TLB size (sequential traffic generator)

Results about the required clock cycles of 4.10				
	Conf. 1	Conf. 2	Conf. 3	Conf. 4
Average (50 th percentile)	23 cc	6 cc	6 cc	6 cc
95 th percentile	431 cc	422 cc	492 cc	492 cc
99 th percentile	643 cc	620 cc	504 cc	504 cc
Minimum value	6 cc	6 cc	6 cc	6 cc
Maximum value	1661 cc	1661 cc	1661 cc	1661 cc

Table 4.11: Time (in cc) results of configurations of Table 4.10, used to explore the performance effects of ATU's L1 TLB size (sequential traffic generator)

Results about the architectural metrics of Table 4.10				
	Conf. 1	Conf. 2	Conf. 3	Conf. 4
One ATU's L1 SMU cache hits	4096	4096	4069	4069
One ATU's L1 SMU cache misses	9	10	10	10
One ATU's L1 TLB cache hits	4096	4096	4096	4069
One ATU's L1 TLB cache misses	60822	23431	17528	17528
ATC's L2 SMU cache hits	0	0	0	0
ATC's L2 SMU cache misses	6	6	6	6
ATC's L2 TLB cache hits	2670	60	1	1
ATC's L2 TLB cache misses	252	92	72	72
HPTW total searches on MM	252	92	72	72
# of 1G PG found by HPTW	88	9	4	4
# of 2M PG found by HPTW	58	12	2	2
# of 4K PG found by HPTW	106	71	66	66

Table 4.12: Architectural results of configurations of Table 4.10 exploring the performance effects of the ATU's L1 TLB size (sequential traffic generator)

By observing the tables mentioned above, one can understand that the performance remains stable using the sequential generator if the L1 TLB size becomes greater than or equal to 8. This is because every ASID's starting VA will be a TLB miss, and every address after that will be a TLB hit (due to the sequence of incoming addresses) - also depending on the position of the randomly chosen starting address on the page, the page size, etc. Thus, as our testbench includes 6 ASIDs in total, a TLB size equal to 8 is enough to store the appropriate PTE for each of these ASIDs. Similar behavior will be observed for the L2 TLB of the ATC, except that the TLB size will now be different. More specifically, L2's TLB size should be greater than or equal to the total number of unique ASIDs communicating with the ATUs connected to the ATC.

4.1.1.3 Performance sensitivity analysis on SMU's FIFO size

Tables 4.13, 4.14, and 4.15 explore the performance fluctuations when the SMU's FIFO size (unique and replay FIFO) changes, and the random traffic generator is utilized.

RV-IOMMU configurations (randomized traffic generator)			
	Conf. 1	Conf. 2	Conf. 3
Number of ATU(s)	1	1	1
Total number of AW Reqs	2048	2048	2048
Total number of AR Reqs	2048	2048	2048
AR Reordering	✓	✓	✓
AW Reordering	✓	✓	✓
ATU Input Buffers	✓	✓	✓
ATU Input Buffers Size	2 entries	2 entries	2 entries
ATU TLB Pipeline Reg	✓	✓	✓
ATC TLB Pipeline Reg	✓	✓	✓
ATU SMU FIFOs size	4	8	16
ATU L1 TLB Size	8	8	8
ATC L2 TLB Size	16	16	16
ATU L1 SMU Cache Associativity	8 Ways	8 Ways	8 Ways
ATU L1 SMU Cache Sets	4	4	4
ATC L2 SMU Cache Associativity	8 Ways	8 Ways	8 Ways
ATC L2 SMU Cache Sets	8	8	8
RAM Read Latency	100cc	100cc	100cc

Table 4.13: RV-IOMMU configurations used to explore the performance effects of the SMU's FIFO sizes (random traffic generator)

Results about the required clock cycles of Table 4.13			
	Conf. 1	Conf. 2	Conf. 3
Average (50 th percentile)	1550 cc	1550 cc	1550 cc
95 th percentile	1962 cc	1962 cc	1962 cc
99 th percentile	2166 cc	2166 cc	2162 cc
Minimum value	423 cc	423 cc	423 cc
Maximum value	3614 cc	3614 cc	3614 cc

Table 4.14: Time (in cc) results of configurations of Table 4.13, used to explore the performance effects of the SMU's FIFO size (random traffic generator)

Results about the architectural metrics of Table 4.13			
	Conf. 1	Conf. 2	Conf. 3
One ATU's L1 SMU cache hits	4096	4096	4069
One ATU's L1 SMU cache misses	10	9	9
One ATU's L1 TLB cache hits	4096	4096	4096
One ATU's L1 TLB cache misses	790022	790020	790020
ATC's L2 SMU cache hits	0	0	0
ATC's L2 SMU cache misses	6	6	6
ATC's L2 TLB cache hits	29	29	29
ATC's L2 TLB cache misses	4064	4064	4064
HPTW total searches on MM	4064	4064	4064
# of 1G PG found by HPTW	1809	1809	1809
# of 2M PG found by HPTW	913	913	913
# of 4K PG found by HPTW	1342	1342	1342

Table 4.15: Architectural measurements of configurations of Table 4.13 exploring the performance effects of the SMU's FIFOs size (random traffic generator)

By observing the tables mentioned above, one can understand that the SMU's FIFO size does not affect our implementation's overall performance as there are only 6 different ASIDs in our tests, which is not a number that can provide a performance fluctuation. The same occurs for the sequential traffic generator as the way that the AXI IDs (potential ASIDs) are generated is the same as the AXI IDs generated by the random traffic generator. However, in theory, the only thing that could be achieved by increasing the SMU's FIFO size would also be to increase the outstanding missing AXI IDs to ASIDs. This, however, would not lead to a performance increase.

4.1.1.4 Performance sensitivity analysis on TLB's FIFO size

Tables 4.16, 4.17, and 4.18 explore the performance fluctuations when the TLB's FIFO size (unique and replay FIFO) changes and the random traffic generator is utilized.

RV-IOMMU configurations (randomized traffic generator)			
	Conf. 1	Conf. 2	Conf. 3
Number of ATU(s)	1	1	1
Total number of AW Reqs	2048	2048	2048
Total number of AR Reqs	2048	2048	2048
AR Reordering	✓	✓	✓
AW Reordering	✓	✓	✓
ATU Input Buffers	✓	✓	✓
ATU Input Buffers Size	2 entries	2 entries	2 entries
ATU TLB Pipeline Reg	✓	✓	✓
ATC TLB Pipeline Reg	✓	✓	✓
ATU TLB FIFOs size	8	16	32
ATU L1 TLB Size	8	8	8
ATC L2 TLB Size	16	16	16
ATU L1 SMU Cache Associativity	8 Ways	8 Ways	8 Ways
ATU L1 SMU Cache Sets	4	4	4
ATC L2 SMU Cache Associativity	8 Ways	8 Ways	8 Ways
ATC L2 SMU Cache Sets	8	8	8
RAM Read Latency	100cc	100cc	100cc

Table 4.16: RV-IOMMU configurations used to explore the performance effects of the TLB's FIFO sizes (random traffic generator)

Results about the required clock cycles of Table 4.16			
	Conf. 1	Conf. 2	Conf. 3
Average (50 th percentile)	1550 cc	1550 cc	1550 cc
95 th percentile	1962 cc	1962 cc	1962 cc
99 th percentile	2166 cc	2166 cc	2162 cc
Minimum value	423 cc	423 cc	423 cc
Maximum value	3614 cc	3614 cc	3614 cc

Table 4.17: Time (in cc) results of configurations of Table 4.16, used to explore the performance effects of the TLB's FIFO size (random traffic generator)

Results about the architectural metrics of Table 4.16			
	Conf. 1	Conf. 2	Conf. 3
One ATU's L1 SMU cache hits	4096	4096	4069
One ATU's L1 SMU cache misses	10	10	10
One ATU's L1 TLB cache hits	4096	4096	4096
One ATU's L1 TLB cache misses	790022	790022	790022
ATC's L2 SMU cache hits	0	0	0
ATC's L2 SMU cache misses	6	6	6
ATC's L2 TLB cache hits	29	29	29
ATC's L2 TLB cache misses	4064	4064	4064
HPTW total searches on MM	4064	4064	4064
# of 1G PG found by HPTW	1809	1809	1809
# of 2M PG found by HPTW	913	913	913
# of 4K PG found by HPTW	1342	1342	1342

Table 4.18: Architectural measurements of configurations of Table 4.16 exploring the performance effects of the SMU's FIFOs size (random traffic generator)

By observing the aforementioned tables, one can understand that the TLB's FIFO size does not affect our implementation's overall performance. This happens, as every new request is – almost always – a TLB miss when we use the random traffic generator. The same also applies when we utilize the sequential traffic generator. The only thing that could happen by increasing the TLB's FIFO size would be to increase the outstanding TLB miss requests accordingly. This, however, would not lead to a performance increase in our test scenarios (traffic generators).

4.1.1.5 Trying to simulate an actual system under a burst

This sub-section will try to identify potential fluctuations of RV-IOMMU's performance in a simulated environment that will mimic a scenario observed in an actual system. Since we do not currently have a representative traffic generator (as we only have two FIFO filled with requests), the only way to stress-test the RV-IOMMU would be through the simulated environment. We will try to mimic the scenario that a new incoming request will always be available until the FIFO becomes empty. To simulate the MM (RAM) delay, we will enforce a 100cc read delay to our RAM. Moreover, we will choose the sequential traffic pattern to match most bursts as in an actual system. We expect that our implementation will be stressed as all the requests for all ATUs will start at the same clock cycle (all FIFOs are ready to feed the corresponding AW or AR of the ATU that is connected at the same clock cycle). This will allow the ATC to accept simultaneous

valid requests from all connected ATUs. In this way, the ATC will choose one of them and stall the remaining. This, however, would be a significant performance bottleneck. Lastly, we will run the configurations shown on 4.19 to change the number of connected ATUs on a unique ATC to observe the performance.

Configurations RV-IOMMU (sequential traffic generator)						
	Conf. 1	Conf. 2	Conf. 3	Conf. 4	Conf. 5	Conf. 6
Number of ATU(s)	1	2	8	32	64	128
Total number of AW Reqs	2048	4096	16384	65536	131072	262144
Total number of AR Reqs	2048	4096	16384	65536	131072	262144
AR Reordering	✓	✓	✓	✓	✓	✓
AW Reordering	✓	✓	✓	✓	✓	✓
ATU Input Buffers	✓	✓	✓	✓	✓	✓
ATU Input Buffers Size	2	2	2	2	2	2
ATU TLB	✓	✓	✓	✓	✓	✓
Pipeline Reg ATC TLB	✓	✓	✓	✓	✓	✓
Pipeline Reg ATU L1 TLB Size	16	16	16	16	16	16
ATC L2 TLB Size	32	32	32	32	32	32
ATU SMU FIFOs size	8	8	8	8	8	8
ATU TLB FIFOs size	8	8	8	8	8	8
ATU L1 SMU Cache Associativity	8	8	8	8	8	8
ATU L1 SMU Cache Sets	4	4	4	4	4	4
ATC L2 SMU Cache Associativity	16 Ways	16 Ways	16 Ways	16 Ways	16 Ways	16 Ways
ATC L2 SMU Cache Sets	16	16	16	16	16	16
RAM Read Latency	100cc	100cc	100cc	100cc	100cc	100cc

Table 4.19: RV-IOMMU configurations used to explore the performance of RV-IOMMU for multiple numbers of connected ATUs feed by sequential traffic generator

Results about the required clock cycles of Table 4.19						
	Conf. 1	Conf. 2	Conf. 3	Conf. 4	Conf. 5	Conf. 6
Average (50 th percentile)	6 cc	6 cc	6 cc	6 cc	6 cc	6 cc
95 th percentile	492 cc	1099 cc	3599 cc	15999 cc	33414 cc	64105 cc
99 th percentile	504 cc	1125 cc	4332 cc	18023 cc	35886 cc	69108 cc
Minimum value	6 cc	6 cc	6 cc	6 cc	6 cc	6 cc
Maximum value	1661 cc	2918 cc	10893 cc	43011 cc	892663 cc	168375 cc

Table 4.20: Table's 4.19 time (in cc) results of configurations. These configurations were used to explore the RV-IOMMU performance when multiple ATUs are connected and fed by the sequential traffic generator.

Results about the architectural metrics of Table 4.19						
	Conf. 1	Conf. 2	Conf. 3	Conf. 4	Conf. 5	Conf. 6
One ATU's L1 SMU cache hits	4096	4096	4069	4069	4069	4069
One ATU's L1 SMU cache misses	10	10	9	12	13	12
One ATU's L1 TLB cache hits	4096	4096	4096	4069	4096	4069
One ATU's L1 TLB cache misses	17631	36818	164069	657673	1329197	2245960
ATC's L2 SMU cache hits	0	8	42	191	391	775
ATC's L2 SMU cache misses	6	6	6	6	6	6
ATC's L2 TLB cache hits	2	14	165	537	987	2353
ATC's L2 TLB cache misses	72	140	577	2323	4546	9064
HPTW total searches on MM	72	140	577	2323	4546	9064
# of 1G PG found by HPTW	4	4	18	61	42	39
# of 2M PG found by HPTW	2	5	26	131	249	520
# of 4K PG found by HPTW	66	131	533	2131	4255	8505

Table 4.21: Architectural metrics of configurations of Table 4.19 used to explore the performance of RV-IOMMU for multiple numbers of connected ATUs feed by a sequential traffic generator

As observed, when we increase the number of connected ATUs on a unique ATC, the tail latency increases accordingly. This is, however, an expected behavior as the valid requests for the ATC (at least at the duration of the first burst) are arriving in a synchronous way (meaning that the ATC will have to serve simultaneous valid requests from all the connected ATUs). As we have already mentioned in Section 3.5.1, the Request Switch, implemented in a round-robin policy, is responsible for choosing a single request at a time. This, however, stalls all the non-served requests increasing the tail latency if new ATUs are added. Nonetheless, we observe that the meantime of serving a request remains at 6cc for every configuration described in Table 4.19. This is a feature of our RV-IOMMU's architect structure and the reason we call it scalable.

4.2 Phase 2: Evaluation on FPGA

4.2.1 Target Platform

The Target Platform is an MPSoC module integrating a Xilinx Zynq UltraScale+. Zynq UltraScale+ MPSoC is the Xilinx second-generation Zynq platform, combining a Processing System (PS) and user-Programmable Logic (PL) into the same device. The Zynq UltraScale+ MPSoC PS block has three major processing units:

- Cortex-A53 application processing unit (APU) - ARM v8 architecture-based 64-bit quad-core multiprocessing CPU.
- Cortex-R5 real-time processing unit (RPU) - ARM v7 architecture-based 32-bit dual real-time processing unit with dedicated tightly coupled memory (TCM).
- Mali-400 graphics processing unit (GPU) with a pixel and geometry processor and a 64KB L2 cache.

On the PL of this board, a user can program the FPGA and generate blocks that the PS could access. This is where the RV-IOMMU is being generated and evaluated.

4.2.2 Implemented Experiment

The design that we implement to evaluate the RV- IOMMU on the FPGA is described in this subsection. In the block design, we use some IPs from Xilinx. The Xilinx IPs we use are:

- AXI Interconnect: to connect the IPs as shown in Figure 4.14.
- AXI CDMA, a Xilinx's implementation of DMA [11]: to create AXI Read and Write requests with low software overhead and to stress the RV-IOMMU.
- AXI BRAM Controller & BRAM: We pre-load the AXI ID to ASID table and a three-level Page Table that translates the incoming addresses from VA 9XXX_XXXX to PA 4XXX_XXXX and supports all possible AXI IDs.
- Virtual Input/Output (VIO): to easily reset all peripherals using Vivado's UI.
- Processor System Reset: to transfer the reset signal around the design.

The aforementioned block design is shown in Figure 4.14. As previously mentioned, we use the Zynq UltraScale+ MPSoC to run a bare-metal C program. We run the C-based code on an ARM processor of the platform using the Vitis IDE. This program consists of two main tasks. The first is to write 4KB to the RAM from the IOMMU's path. After that, to check their validity through both a non-IOMMU path and an IOMMU one. The second one is to check if the RV-IOMMU

can handle more demanding situations by stress testing it. To accomplish that, we needed to observe whether the RV-IOMMU could switch between a ready to a not-ready state and then back to the former. By doing that, one can understand that the RV-IOMMU can recover from a not-ready state after a short period of time. To generate situations like this, we configure a DMA engine to generate requests that will be handled from the RV-IOMMU. The DMA engine is needed to feed our RV-IOMMU as, in this way, the overhead/time penalty of passing a request from PS to PL is minimized. The CDMA is programmed by writing specific addresses referring to its control registers. As indicated in [11], this process is part of the C-program's second task mentioned above. In this way, the CDMA is programmed to transfer 2KB of data. Both of the read and write addresses are referred to as virtual addresses allowing the RV-IOMMU to translate them. The C-program's last step is to validate that the new chunk of memory written by the CDMA contains the correct/new values. It is important to mention here that the same experiment was executed with both AXI4 compatible configurations, i.e., with and without input buffers on ATU.

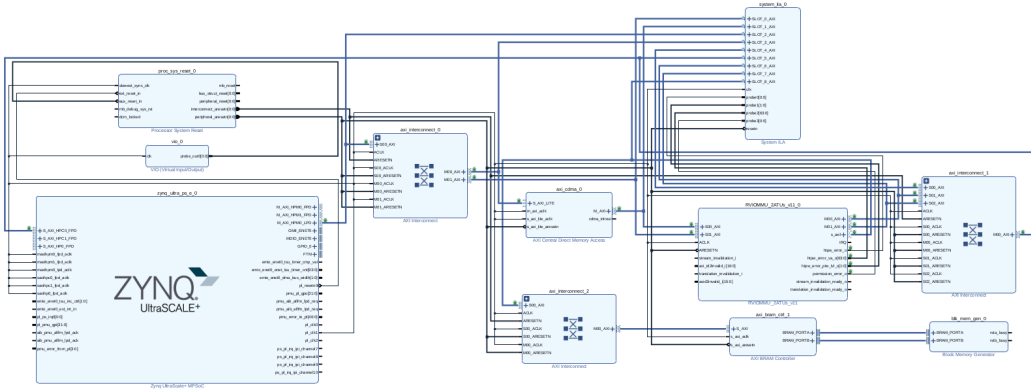


Figure 4.14: Implemented block design of FPGA evaluation

4.2.3 Timing Requirements

This subsection provides the report of the timing requirements of the RV-IOMMU for different configurations. All these metrics are generated using the Synthesis function (with Flow_PerfOptimized_high strategy) of Vivado 2020.1. Since we use only the Synthesis function of Vivado and not the implementation (where I/O errors occur due to the limitation of available pins), these metrics are pessimistic.

The effect of enabling the input buffers (FIFOs) at ATU regarding the timing requirements for different configurations is collected and presented in Table 4.22.

4.2.4 Synthesis Utilization

This subsection provides the FPGA HW sources report needed to generate an RV-IOMMU for three different configurations. All there metrics were generated by using the Synthesis function (with Flow_PerfOptimized_high strategy) of Vivado 2020.1.

The first examined configuration had the following parameters:

- One ATU & one ATC
- No ATU input buffers
- No Reordering
- ATU: SMU FIFO size (Replay and Unique) = 4
- ATU: TLB FIFO size (Replay and Unique) = 4
- ATC: SMU Invalidation FIFO size = 4
- ATC: TLB Invalidation FIFO size = 4
- ATC: SMU Response FIFO size = 4
- ATC: TLB Response FIFO size = 4
- ATC: SMU Invalidation FIFO size = 4
- ATU: SMU Set-associative Cache (Level 1): Associativity = 8 & Cache Sets = 8
- ATC: SMU Set-associative Cache (Level 2): Associativity = 16 & Cache Sets = 16
- ATU: Level 1 TLB size = 16
- ATC: Level 2 TLB size = 64

Based on the above, the hardware sources are presented in Table 4.23.

RV-IOMMU with one ATU and one ATC				
Configurable Block	Logic	Total Number	ATU's	ATC's
LUT as logic		20754	6161	14509
Registers as Flip Flop		37883	10533	27350
CARRY8		14	14	0
F7 Muxes		2585	532	2053
F8 Muxes		1013	0	1013

Table 4.23: HW resources for RV-IOMMU with one ATU and one ATC

The second configuration is almost identical to the first, except that we chose to apply the reordering feature on both AW and AR channels. Table 4.24 shows the HW requirements of this second configuration. In this way, we manage to estimate the additional HW resources of the reordering feature. Based on Table 4.23 and Table 4.24, this configuration's reordering feature needs an extra 1561 Lookup Tables LUTs as logic, 76 registers as Flip-Flops, and 20 F7 Muxes. While the reordering feature reflects only on the ATU, the above numbers can be interpreted as 25% more ATU LUTs as logic, 0.007% more registers as Flip-Flops, and 0.037% more F7 Muxes (the percentages refer only to ATU metrics).

RV-IOMMU with one ATU and one ATC				
Configurable Block	Logic	Total Number	ATU's	ATC's
LUT as logic		22315	7723	14508
Registers as Flip Flop		37959	10609	27350
CARRY8		14	14	0
F7 Muxes		2605	552	2053
F8 Muxes		1013	0	1013

Table 4.24: HW resources for RV-IOMMU with one ATU and one ATC with ATU's reordering feature

The third and last configuration that we use is almost identical to the second with the addition of two input FIFOs (buffers) of size 8 on ATU incoming channels. Table 4.25 shows the HW requirements of this configuration. This allows us to estimate the additional HW resources for the reordering feature and input buffers. By comparing 4.24 to Table 4.25, one can observe that the input buffers with the reordering feature need an extra 286 LUTs as logic, 1599 registers as Flip-Flops, and 97 F7 Muxes, as opposed to the second configuration. While the input buffers are on the ATU, the above numbers can be interpreted as approximately 0.037% more ATU LUTs as logic, 0.15% more register as Flip-Flops, and 0.175% more F7 Muxes (the percentages refer only to ATU metrics).

RV-IOMMU with one ATU and one ATC				
Configurable Block	Logic	Total Number	ATU's	ATC's
LUT as logic		22601	8190	14327
Registers as Flip Flop		39558	12208	27350
CARRY8		14	14	0
F7 Muxes		2702	649	2053
F8 Muxes		1013	0	1013

Table 4.25: HW resources for RV-IOMMU with one ATU and one ATC with ATU's reordering feature and input buffers

4.2.5 Implementation Utilization

This subsection provides the report of the timing requirements of the RV-IOMMU for the configuration that we evaluate on the Ultrascale+ FPGA (see 4.2). All these metrics were generated using the Implementation function (with Flow_PerfOptimized_high strategy) of Vivado 2020.1.

The examined configuration had the following parameters:

- Two ATUs & one ATC
- Enabled input buffers with size = 8
- Enforce Reordering on both AW & AW channels
- ATU: SMU FIFO size (Replay and Unique) = 4
- ATU: TLB FIFO size (Replay and Unique) = 4
- ATC: SMU Invalidation FIFO size = 4
- ATC: TLB Invalidation FIFO size = 4
- ATC: SMU Response FIFO size = 4
- ATC: TLB Response FIFO size = 4
- ATC: SMU Invalidation FIFO size = 4
- ATU: SMU Set-associative Cache (Level 1): Associativity = 8 & Cache Sets = 8
- ATC: SMU Set-associative Cache (Level 2): Associativity = 16 & Cache Sets = 16
- ATU: Level 1 TLB size = 16
- ATC: Level 2 TLB size = 64

Based on the above, the hardware resources are presented in Table 4.26. The Configurable Logic Block (CLB) is the main resource for implementing general-purpose combinatorial and sequential circuits. Every CLB contains one slice with eight 6-input LUTs and sixteen Flip Flops (storage elements). One LUT can be configured as (a) a 6-input LUT with one output or (b) two 5-input LUTs with separate outputs but common addresses or logic inputs. The available FPGA resources are 274080 CLB LUTs and 548160 CLB Registers.

FPGA implemented RV-IOMMU with two ATUs and one ATC (4.2)				
Configurable Block	Logic	Total Number	one ATU's	one ATC's
CLB LUTs		8016	3130 or 3000	1885
CLB Registers		12447	5095	2256
CARRY8		44	22	0
F7 Muxes		122	26	70
F8 Muxes		0	0	0

Table 4.26: Actual HW resources for RV-IOMMU with two ATU and one ATC implemented on the FPGA

In this way, we figure out that the instance mentioned above of RV-IOMMU that we also use for the evaluation on FPGA (see 4.2) uses 2.9% of the available CLB LUTs and 2.3% of the available CLB Registers.

Table 4.27 provides a comparison of the required HW resources comparing the RV-IOMMU (with 2 ATUs and 1 ATC), the Ariane core [6], and the CDMA [11]. All these metrics were generated using the Implementation function of Vivado 2020.1 with the target platform a Zynq UltraScale+ MPSoC ZU9CG (part number: xczu9eg-ffvc900-2-e) [12].

Comparison of used HW resources				
Configurable Block	Logic	RV-IOMMU (2 ATUs, 1 ATC)	Ariane core	CDMA
CLB LUTs		8016	39855	949
CLB Registers		12447	21373	1444
CARRY8		44	676	14
F7 Muxes		122	2414	10
F8 Muxes		0	181	0

Table 4.27: Actual HW resources comparison for (a) RV-IOMMU with two ATU and one ATC, (b) Ariane core, and (c) CDMA - all implemented on the same FPGA

Chapter 5

Conclusion and future work

5.1 Summary

This thesis contributes to multiple aspects of the hardware design for RISC-V IOMMUs. First of all, we design the architecture of a scalable IOMMU for RISC-V Architectures. Then we implement the RV-IOMMU in SystemVerilog following a parametric approach and develop many user-defined features such as reordering and controlling the number of generated ATUs. According to the 64-bit RISC-V ISA, our RV-IOMMU is compatible with Sv39 and Sv48. We also evaluate our implementation using simulation. Lastly, we evaluate the RV-IOMMU on FPGA by creating a block design that includes RV-IOMMU in order to verify our design. To conclude, we implement an HW component that accepts read and write AXI requests and, if allowed, performs the address translation from Virtual (included in the initial incoming request) to Physical (included at the outcome request of the RV-IOMMU) addresses.

5.2 Future Work

Some logic of the implementation was not implemented optimally due to time constraints. As a next step, efforts will focus on implementing the modules listed below:

- A more sophisticated way to handle the incoming request that does not have the permission to accomplish the action.
- Improve the TLB input controller with an FSM (similar to the SMU's one).
- Replace the control logic of the RV-IOMMU with a credit-based one.

Bibliography

- [1] Dynamic dma mapping using the generic device. <https://www.kernel.org/doc/Documentation/DMA-API.txt>.
- [2] Ieee standard for systemverilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, 2018.
- [3] Advanced microcontroller bus architecture - amba. <https://developer.arm.com/architectures/system-architectures/amba>, 2021.
- [4] Amba axi and ace protocol specification. <https://developer.arm.com/documentation/ih10022/h>, 2021.
- [5] Arm system memory management unit architecture specification - smmu architecture version 2.0. <https://developer.arm.com/documentation/ih10062/dc/>, 2021.
- [6] The cva6 (formerly ariane) is an application class 6-stage risc-v cpu capable of booting linux. <https://github.com/openhwgroup/cva6>, 2021.
- [7] Input-output memory management unit. https://en.wikipedia.org/wiki/Input%E2%80%93output_memory_management_unit, 2021.
- [8] Kernel. [https://en.wikipedia.org/wiki/Kernel_\(operating_system\)](https://en.wikipedia.org/wiki/Kernel_(operating_system)), 2021.
- [9] Physical address extension. https://en.wikipedia.org/wiki/Physical_Address_Extension, 2021.
- [10] The risc-v instruction set manual volume ii: Privileged architecture document version 1.12-draft. <https://riscv.org/technical/specifications/privileged-isa/>, 2021.
- [11] Axi central direct memory access v4.1. https://www.xilinx.com/support/documentation/ip_documentation/axi_cdma/v4_1/pg034-axi-cdma.pdf, April 4, 2018.

- [12] Zynq ultrascale+ device: Technical reference manual. https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf, December 4, 2020.
- [13] Virtual memory. https://en.wikipedia.org/wiki/Virtual_memory, February 2021.
- [14] Jonathan Corbet. Linux device drivers, chapter 15: Memory mapping and dma. 3rd edition, 2005.
- [15] Jimi Xenidis Muli Ben-Yehuda. The price of safety: Evaluating iommu performance. pages 9–20, 2007.
- [16] M. J. Lanigan T. Kilburn, D. B. G. Edwards and F. H. Sumner. One-level storage system. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, EC-11(2):223–235, April 1962.
- [17] Andrew Waterman and Krste Asanovic. The risc-v instruction set manual, volume i: Unprivileged isa, document version 20191213. *RISC-V Foundation*, 2019.

Appendix A

ARM's Compressed StreamID indexing matching algorithm

Following the Compressed StreamID indexing matching algorithm of ARM's System MMU v2.

- Assume that $\text{StreamID} = \text{strm_id}$
- $\text{Column} = \text{strm_id} \bmod 4$ (SMMU_COMPINDEXn registers size equals to 4 bytes)
- $\text{Row} = \text{strm_id} \text{ DIV } 4$
- The column byte of SMMU_COMPINDEXrow holds the value of the SMMU_S2CRn for the stream.

This array's total size is 64KB and is calculated if we think that the StreamId is up to 16-bits to support 64K different StreamIDs. There are 16K rows at the array. As a result, this array's total size is $16\text{K} * 4 \text{ B} = 64\text{KB}$ (one byte per StreamID).

ctrl + s , alt + F4