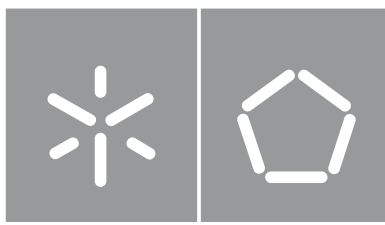


Universidade do Minho
Escola de Engenharia

Manuel Alejandro Cedeño Rodríguez

**An Open-source RISC-V Input/Output
Memory Management Unit**



Universidade do Minho
Escola de Engenharia

Manuel Alejandro Cedeño Rodríguez

An Open-source RISC-V Input/Output Memory Management Unit

Dissertação de Mestrado
Mestrado em Engenharia Eletrónica
Industrial e Computadores
Sistemas Embebidos e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Sandro Pinto

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho



**Atribuição
CC BY**

<https://creativecommons.org/licenses/by/4.0/>

Agradecimentos

Primeiramente, quero agradecer a Deus por me ter guiado durante estes seis anos de mudanças e desenvolvimento. Por me ter concedido a capacidade de concluir este curso com sucesso e ultrapassar todos os desafios que surgiram a nível pessoal e profissional. Por ter colocado no meu caminho pessoas maravilhosas que foram fundamentais para o meu crescimento em todos os sentidos.

Quero agradecer aos meus pais, Maria e Emmanuel, por terem moldado a pessoa que sou hoje. Por me terem educado com amor e responsabilidade, transmitindo valores dignos. Pelo apoio incondicional, amor e suporte ao longo da minha vida. Vocês representam um pilar inabalável no meu ser, e a minha principal motivação para continuar na luta do dia-a-dia. Às minhas irmãs, Yenisey e Yemdri, por me acompanharem e apoiarem ao longo do meu percurso. Por estarem sempre determinadas a ajudar-me, a motivar-me e a aplaudir todas as minhas conquistas. Este sucesso também é vosso.

Agradeço à minha família em geral, em Portugal e na Venezuela. Àqueles que tornaram possível a concretização do meu desejo de emigrar através de qualquer tipo de ajuda. Às pessoas que Deus colocou no meu caminho ao longo destes seis anos, e que contribuíram para o meu desenvolvimento pessoal e profissional. Aos colegas de trabalho que se tornaram amigos de valor incalculável. Aos meus colegas de curso, pela companhia e contributo ao longo do caminho. A todos vós, o meu muito obrigado.

Quero expressar a minha gratidão a todas as pessoas que me apoiaram, de alguma forma, em qualquer fase do meu percurso. A todos aqueles que acreditaram fielmente em mim, demonstrando orgulho e felicidade pelas minhas conquistas. A todos os que me ajudaram e me motivaram a continuar em momentos de grande dificuldade, dando uma mão, uma palavra de encorajamento ou simplesmente estando lá quando eu mais precisei. Mesmo os mais resilientes precisam de apoio nestas situações.

Por fim, faço um agradecimento especial ao meu irmão, Joan Manuel, que foi o meu companheiro de luta, o meu melhor amigo, e é agora o meu anjo da guarda. Não tivemos a oportunidade de celebrar esta vitória juntos, em pessoa. No entanto, sei que do céu estás extremamente orgulhoso de mim e que sorris e comemoras cada conquista que alcanço. Este trabalho, assim como todos os triunfos da minha vida, são especialmente dedicados a ti.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

Uma Unidade de Gestão de Memória de Entrada/Saída RISC-V de Código Aberto

A constante procura por melhor desempenho e funcionalidades no software levou ao desenvolvimento de arquiteturas de hardware e plataformas mais complexas. Estes sistemas integram vários dispositivos com capacidade de acesso direto à memória (DMA), usados para libertar o processador da gestão destes acessos. No entanto, estes mecanismos podem introduzir problemas de segurança, criando a possibilidade de ataques por meio de periféricos maliciosos. Para lidar com estes problemas, os computadores modernos integram hardware adicional ao nível do sistema, nomeadamente, a Unidade de Gestão de Memória de Entrada/Saída (IOMMU). Uma IOMMU atua como intermediária na gestão dos acessos dos dispositivos à memória, aplicando verificação de permissões e tradução de endereços.

Atividades de investigação e desenvolvimento na academia deram origem a RISC-V, uma arquitetura de conjunto de instruções que segue uma filosofia completamente aberta. Recentemente, a comunidade RISC-V desenvolveu a especificação RISC-V IOMMU v1.0, e, até ao momento, não foi implementada nenhuma Propriedade Intelectual (IP) RISC-V IOMMU de código aberto. Além disso, no contexto da IOMMU RISC-V, nenhum trabalho anterior realizou (i) um estudo exploratório relativamente às características arquiteturais e configurações microarquiteturais da IOMMU, e (ii) uma avaliação empírica sobre os custos de hardware e as latências de tradução.

Esta dissertação de mestrado descreve a implementação do primeiro IP IOMMU de código aberto compatível com a especificação RISC-V IOMMU v1.0. O IP foi integrado num *System-on-Chip* com suporte para virtualização, baseado no processador CVA6. Após a validação do IP, analisámos o impacto funcional e no uso de recursos de várias configurações de hardware, com base em diferentes combinações de funcionalidades arquiteturais e propriedades microarquiteturais da IOMMU. No processo de avaliação, usámos várias configurações de software para avaliar o IP em termos de custos de hardware e latências de tradução. Por fim, discutimos *designs* alternativos e modificações arquiteturais/microarquiteturais para melhorar o IP e a especificação RISC-V IOMMU. Todos os artefactos relacionados com este trabalho foram disponibilizados em código aberto para promover a colaboração na comunidade RISC-V.

Palavras-chave: Arquitetura, FPGA, IOMMU, RISC-V, Virtualização.

Abstract

An Open-source RISC-V Input/Output Memory Management Unit

The increasing desire for software performance and features have led to more complex hardware architectures and computing platforms. These systems integrate multiple bus masters equipped with *Direct Memory Access* (DMA) engines, used to alleviate processors from memory access management. However, DMA-capable devices may open security holes that attackers can leverage to bypass CPU memory protection. To tackle this attack vector, computing units started integrating specific system-level hardware controllers, like the *Input/Output Memory Management Unit* (IOMMU). An IOMMU mediates and manages device accesses to memory, enforcing permission checks and address translation.

Research and development efforts among academia led to the rise of RISC-V, an open-standard *Instruction Set Architecture* (ISA). In the last few years, the RISC-V community has been working on the IOMMU specification, recently ratified as v1.0. To the best of our knowledge, no open-source RISC-V IOMMU *Intellectual Property* (IP) has been developed at the time of writing. Furthermore, in the context of the RISC-V IOMMU, no previous work has performed (i) a design space exploration concerning its architectural features and microarchitectural configurations and (ii) an empirical evaluation to assess hardware costs and additional latencies incurred by address translation.

In this Master Thesis, we implemented the first open-source IOMMU IP fully compliant with the RISC-V IOMMU specification v1.0. Subsequently, we integrated the IP into a single-core CVA6-based System-on-Chip with virtualization support. We conducted a design space exploration to analyze the functional and resource-utilization impact of various hardware configurations based on different combinations of IOMMU architectural features and microarchitectural properties. In the evaluation process, we used various software setups (e.g., baremetal, Linux, and Bao hypervisor) to assess the IOMMU IP regarding hardware costs and translation latencies. Finally, we discussed alternative designs and architectural/microarchitectural modifications to enhance the implemented IP and the overall IOMMU specification. All the artifacts related to this work were open-sourced to foster collaboration within the RISC-V community.

Keywords: Architecture, FPGA, IOMMU, RISC-V, Virtualization.

Table of Contents

1	Introduction	1
1.1	Motivation	2
1.2	Goals	3
1.3	Document Structure	3
2	Background and State of the Art	5
2.1	The Basics of Computer Architecture	5
2.2	RISC-V	6
2.2.1	RISC-V Cores	8
2.3	Virtual Memory	10
2.4	Virtualization	12
2.4.1	Hypervisor Architectures	13
2.4.2	RISC-V Hypervisor Extension	15
2.4.3	IO Virtualization	17
2.5	Direct Memory Access	18
2.5.1	DMA as a Potential Threat	18
2.6	Related Work	19
2.6.1	Custom IOMMU-related Solutions	19
2.6.2	Intel VT-d	22
2.6.3	AMD-Vi	24
2.6.4	ARM SMMU	26
2.6.5	An Overview of the RISC-V IOMMU	27
2.6.6	IOMMU Comparison and Gap Analysis	28

3	RISC-V IOMMU Specification	32
3.1	Address Translation	32
3.1.1	IO Address Translation Caches	33
3.2	MSI Translation	34
3.3	Software Interface	35
3.3.1	Memory-mapped Register Interface	35
3.3.2	In-memory Queue Interface	35
3.3.3	IOMMU Interrupt Generation	37
3.4	Miscellaneous Features	37
3.4.1	PCIe ATS and PRI	38
3.4.2	Hardware Performance Monitor	38
3.4.3	Register-based Debug Interface	39
3.5	Use Cases for Virtualization	39
4	RISC-V IOMMU Design	41
4.1	Architectural Requirements	41
4.2	Design Parameters	42
4.3	Base Microarchitecture	43
4.4	Translation Logic	43
4.4.1	Context Directory Walker	44
4.4.2	Page Table Walker	48
4.4.3	IO Address Translation Caches	50
4.5	Software Interface	51
4.5.1	Memory-mapped Register Interface	52
4.5.2	Command Queue Handler	54
4.5.3	Fault Queue Handler	55
4.5.4	Interrupt Generation Support	56
4.5.5	Hardware Performance Monitor	57
4.6	External Interfaces	58
4.6.1	Programming Interface	58
4.6.2	Data Structures Interface	59
4.6.3	Translation Request and Completion Interfaces	59

5	RISC-V IOMMU Implementation	62
5.1	Key Implementation Details	62
5.1.1	Parameterizable Implementation	63
5.1.2	The Finite State Machine Model	65
5.1.3	IOMMU Packages	67
5.1.4	Third-party Resources	69
5.2	RISC-V IOMMU Driver for Bao Hypervisor	70
5.2.1	Bao Data Structuring	70
5.2.2	IOMMU Driver	72
6	Integration, Validation, and Evaluation	75
6.1	Platform-level Integration	75
6.2	Functional Validation	77
6.2.1	Custom Test Framework	78
6.2.2	System-level Validation	80
6.3	Evaluation	84
6.3.1	Hardware Resources	85
6.3.2	Translation Latencies	86
7	Conclusion	89
7.1	Future Work	90
	References	92

Acronyms

AIA	<i>Advanced Interrupt Architecture.</i>
ALU	<i>Arithmetic and Logic Unit.</i>
API	<i>Application Programming Interface.</i>
APLIC	<i>Advanced Platform-Level Interrupt Controller.</i>
ASIC	<i>Application-Specific Integrated Circuit.</i>
ASID	<i>Address Space ID.</i>
ATS	<i>Address Translation Services.</i>
CDW	<i>Context Directory Walker.</i>
CISC	<i>Complex Instruction Set Computer.</i>
COTS	<i>Commercial Off-The-Shelf.</i>
CPS	<i>Cyber-Physical Systems.</i>
CPU	<i>Central Processing Unit.</i>
CQ	<i>Command Queue.</i>
CSR	<i>Control and Status Register.</i>
DC	<i>Device Context.</i>
DDT	<i>Device Directory Table.</i>
DDTC	<i>Device Directory Table Cache.</i>
DMA	<i>Direct Memory Access.</i>
FF	<i>Flip-Flop.</i>
FIFO	<i>First In First Out.</i>
FPGA	<i>Field Programmable Gate Array.</i>
FQ	<i>Fault/Event Queue.</i>

FSM	<i>Finite State Machine.</i>
GPA	<i>Guest Physical Addresses.</i>
GSCID	<i>Guest Soft-Context Identifier.</i>
GVA	<i>Guest Virtual Address.</i>
HDD	<i>Hard-Disk Drive.</i>
HDL	<i>Hardware Description Language.</i>
HPM	<i>Hardware Performance Monitor.</i>
I/O	<i>Input/Output.</i>
IGS	<i>Interrupt Generation Support.</i>
IMSIC	<i>Incoming Message-Signaled Interrupt Controller.</i>
IOATC	<i>IO Address Translation Cache.</i>
IOMMU	<i>Input/Output Memory Management Unit.</i>
IOMPU	<i>IO Memory Protection Unit.</i>
IOTLB	<i>I/O Translation Look-aside Buffer.</i>
IOVA	<i>IO Virtual Address.</i>
IP	<i>Intellectual Property.</i>
ISA	<i>Instruction Set Architecture.</i>
ISR	<i>Interrupt Service Routine.</i>
LUT	<i>Look-Up Table.</i>
MMU	<i>Memory Management Unit.</i>
MRIF	<i>Memory-Resident Interrupt File.</i>
MSI	<i>Message-Signaled Interrupt.</i>
OS	<i>Operating System.</i>
PASID	<i>Process Address Space ID.</i>
PC	<i>Process Context.</i>
PCIe	<i>Peripheral Component Interconnect Express.</i>
PDT	<i>Process Directory Table.</i>
PDTC	<i>Process Directory Table Cache.</i>
PIPT	<i>Physically-Indexed-Physically-Tagged.</i>

PLIC	<i>Platform-Level Interrupt Controller.</i>
PQ	<i>Page-Request Queue.</i>
PRI	<i>Page Request Interface.</i>
PSCID	<i>Process Soft-Context Identifier.</i>
PT	<i>Page Table.</i>
PTE	<i>Page Table Entry.</i>
PTW	<i>Page Table Walker.</i>
RISC	<i>Reduced Instruction Set Computer.</i>
RTL	<i>Register-Transfer Level.</i>
RTOS	<i>Real-Time Operating System.</i>
SoC	<i>System-on-Chip.</i>
SPA	<i>Supervisor Physical Address.</i>
SSD	<i>Solid-State Drive.</i>
SWaP-C	<i>Size, Weight, Power and Cost.</i>
TLB	<i>Translation Look-aside Buffer.</i>
VAS	<i>Virtual Address Space.</i>
VIPT	<i>Virtually-Indexed-Physically-Tagged.</i>
VIVT	<i>Virtually-Indexed-Virtually-Tagged.</i>
VM	<i>Virtual Machine.</i>
VMID	<i>Virtual Machine ID.</i>
VMM	<i>Virtual Machine Monitor.</i>
WSI	<i>Wire-Signaled Interrupt.</i>

List of Figures

2.1	CVA6 Core Microarchitecture	9
2.2	Illustration of the Memory Hierarchy Concept	11
2.3	Type-1; Type-2 and Type-1.5 Hypervisors	14
2.4	RISC-V Two-stage Address Translation Flow	16
2.5	Intel VT-d Nested Translation with 4KB Pages	23
2.6	AMD-Vi Command Buffer	25
2.7	ARM SMMU Context Data Structures	26
3.8	RISC-V IOMMU Device Context Formats	33
3.9	MSI Translation Modes Supported by the RISC-V IOMMU	34
3.10	RISC-V IOMMU Memory-based Queue Model	37
3.11	Usage Models of the RISC-V IOMMU in Virtualized Systems	40
4.12	Base IOMMU Microarchitecture	44
4.13	IO Virtual Address Translation Process	45
4.14	Context Directory Walk Procedure	46
4.15	Finite State Machine of the Context Directory Walker	47
4.16	Sv39x4 Nested Memory Walk	49
4.17	Page Table Walker's Finite State Machine	50
4.18	Structure of an Entry in each IOATC	51
4.19	Core Design of the IOMMU Register Map	52
4.20	IOMMU Register Field Design	53
4.21	States Diagram of the Command Queue Handler	54
4.22	Fault Queue Handler's States Diagram	55
4.23	Interrupt Generation Subsystem Design	56
4.24	Hardware Performance Monitor Design	57
4.25	Programming Interface	59

4.26	Data Structures Interface	60
4.27	Translation Request and Completion Interfaces	61
5.28	Bao Data Structures Related to the RISC-V IOMMU Driver	71
6.29	CVA6-based SoC Used in the Validation and Evaluation Setup	76
6.30	Software Stack Used for System-level Validation	80
6.31	Average Latency of 1000 DMA Transfers in the Baremetal Setup	87
6.32	Average Transfer Time of 1000 DMA Transfers in the System-level Software Setups	88

List of Tables

2.1	RISC-V Base ISAs and Standard Extensions	7
2.2	RISC-V Privilege Modes	8
2.3	RISC-V Privilege Modes with the Hypervisor Extension	15
2.4	Architectural Comparison Between the RISC-V IOMMU and Alternative IOMMUs	29
2.5	Features and Status of Existing and Ongoing RISC-V IOMMU Implementations	30
3.6	RISC-V IOMMU Memory-mapped Register Interface	36
3.7	RISC-V IOMMU Standard Commands	38
3.8	RISC-V IOMMU Standard HPM Events	39
4.9	Architectural IOMMU Features Considered for the Design	41
4.10	IOMMU Design Parameters	43
5.11	Third-party Standalone Modules Used to Implement the IOMMU IP	69
6.12	Effect of the <i>InclDMA</i> and <i>InclIOMMU</i> Parameters	78
6.13	Hardware Resources Used by the Target SoC with Different IOMMU Configurations	86

Code Listings

5.1	Instantiation of the WSI Generation Module	63
5.2	Internal Logic of the IOMMU WSI Generation Module	64
5.3	Instantiation of the <i>icvec</i> Register	64
5.4	FQ Handler module: State Encodings and Default Values	66
5.5	FQ Handler Module: <i>WRITE</i> State	66
5.6	FQ Handler Module: Sequential Logic	67
5.7	Processing of the <i>IODIR</i> Command	68
5.8	Implementation of the <i>iommu_arch_init</i> Interface Function	72
5.9	Partial Implementation of the <i>rv_iommu_init</i> Function	72
5.10	Implementation of the <i>iommu_vm_arch_add</i> Function	73
6.11	Test Battery Implemented for the Baremetal Framework	78
6.12	Execution of the <i>two_stage_translation</i> and <i>msi_generation</i> Tests	79
6.13	Linux Boot Messages Related to the IOMMU and iDMA Modules	81
6.14	Output Messages of Two DMA Transfers in the Linux-based Setup	82
6.15	HPM Counters State After Validation in the Linux-based Setup	82
6.16	VM Configuration Used for Bao in the Virtualized Software Setup	83
6.17	Output Messages of Two DMA Transfers in the Virtualization-based Setup	84

Chapter 1

Introduction

Embedded systems serve a broad spectrum of market segments, ranging from general-purpose applications to real-time and safety-critical environments. Some examples are automotive, manufacturing, medical devices, military, traffic control, and safety. Traditionally, these systems were single-purposed, relying on naive architectures tailored to their unique demands. However, as time has progressed, the scope of embedded systems has expanded significantly [1]. Modern embedded systems integrate small subsystems with different functions and levels of criticality, working together to meet specific requirements [2]. For example, *Cyber-Physical Systems* (CPS) seamlessly integrate software and hardware components to perform well-defined tasks, maximizing synergies and achieving significant benefits in *Size, Weight, Power and Cost* (SWaP-C).

This shift towards integration in embedded systems has led to sophisticated hardware architectures and advanced computing platforms [3], incorporating multiple bus masters endowed with memory access capabilities [4, 5]. Notably, DMA engines empower peripherals to directly issue read/write requests to system memory (i.e., without going through the *Central Processing Unit* (CPU)), relieving the processor from the workload associated with managing these memory accesses. Nonetheless, DMA-like engines may introduce security vulnerabilities as they create opportunities for attackers to compromise the system and its hosted applications. Because DMA-capable devices can access arbitrary memory locations, attackers can exploit malicious peripherals to bypass CPU memory protection and isolation mechanisms.

As a step towards tackling DMA-related attack vectors, modern computing units started to be enhanced with specific system-level hardware controllers that interpose between bus masters and memory, in particular, the IOMMU. This module was originally introduced to increase the addressability of a device or bus [6]. The uses of IOMMUs were later extended to restrict the host memory pages that a device can access, enforcing permission checks and address translation on every DMA request. Therefore, IOMMUs provide an increased level of spatial isolation, protecting the system from faulty or malicious device drivers.

Today, numerous processor design and manufacturing companies, including prominent names like Intel, AMD, and ARM, have developed their specifications for IOMMUs [7, 8, 9].

RISC-V [10] sets itself apart from the established classical mainstream ISAs by providing an open standard ISA. This unique ISA incorporates a modular and adaptable extension mechanism, enabling it to scale seamlessly from compact microcontrollers to high-performance supercomputers. In the last few years, the RISC-V community has been working towards developing the RISC-V IOMMU specification, which has recently been ratified as v1.0 [11]. While some proprietary implementations of the RISC-V IOMMU (e.g., SiFive, Ventana) have emerged during the development of the specification, to the best of our knowledge, no open-source RISC-V IOMMU IP has been developed at the time of writing. Furthermore, within the context of the RISC-V IOMMU, no previous work has performed (i) a design space exploration concerning the architectural features and microarchitectural configurations of the IOMMU, and (ii) an empirical evaluation to assess hardware costs and translation latencies.

This Master Thesis describes the design and implementation of the first open-source IOMMU IP fully compliant with the RISC-V IOMMU standard specification v1.0. The IP is subsequently integrated into a single-core CVA6-based *System-on-Chip* (SoC) endowed with virtualization support for validation and evaluation. Furthermore, we analyze the functional and resource-utilization impact of various hardware configurations based on different combinations of IOMMU architectural features and microarchitectural properties. For each configuration, we assess the IOMMU IP regarding hardware costs and translation latencies. Finally, based on the knowledge and experience gained during this research, we discuss alternative designs and architectural/microarchitectural modifications to enhance the implemented IP and the underlying RISC-V IOMMU specification. All artifacts (i.e., IOMMU IP, reference designs, and experiments) are publicly available with permissive licenses to foster collaboration and enable contributions with further design configurations and software stacks.

1.1 Motivation

Unprotected access to physical memory through DMA-capable peripherals represents a significant system-wide vulnerability. In cloud-based environments, such vulnerabilities can result in the exposure of sensitive data, while in more critical domains (e.g., medical, automotive, industrial safety, aeronautics), they can lead to a complete loss of system control, resulting in instability and even endangering lives [12]. An IOMMU effectively addresses this issue by performing address translation and permission checks for every DMA request, significantly enhancing the safety, security, and reliability of computer systems.

Notwithstanding the extra processing overhead introduced by address translation, an IOMMU brings more advantages beyond memory protection. In a virtualized system, a hypervisor can safely assign a physical device exclusively to a guest *Operating System* (OS) running on a *Virtual Machine* (VM). The IOMMU ensures that the guest OS cannot compromise the integrity of other VMs using the assigned device. This capability eliminates the performance overhead typically suffered when the hypervisor virtualizes DMA-capable peripherals to guarantee safe accesses, allowing for near-native performance on the VM.

1.2 Goals

This thesis represents a contribution to computer engineering and the RISC-V community. Our implementation is the first open-source IOMMU IP compliant with the RISC-V IOMMU standard specification v1.0, representing a proof-of-concept and a reference point for further improvement. Furthermore, we leverage our implementation to conduct further research aimed at enhancing the IP and the specification. To achieve these contributions, the following goals were defined for this thesis:

1. Implement an open-source IOMMU IP fully compliant with the RISC-V IOMMU standard specification, and integrate the IP into a single-core CVA6-based SoC with support for virtualization.
2. Add RISC-V IOMMU support in Bao hypervisor [13], enabling the safe assignment of DMA-capable devices to VMs in RISC-V platforms.
3. Analyze the impact of the IOMMU's various architectural features (mandatory vs. optional) and microarchitectural properties regarding translation latencies and IP hardware resource utilization.
4. Identify and discuss alternative designs and architectural and microarchitectural modifications to further optimize and enhance the IOMMU IP and the underlying RISC-V IOMMU specification.

1.3 Document Structure

The remaining chapters of the dissertation are organized as follows:

- Chapter 2 compiles fundamental concepts on the basics of computer architecture, virtualization, DMA, and other context information. In addition, it summarizes state-of-the-art works developed in the context of secure and efficient *Input/Output* (I/O) virtualization, including custom solutions and

proprietary IOMMUs developed by leading semiconductor companies. Lastly, we present existing and ongoing efforts in the context of the RISC-V IOMMU.

- Chapter 3 outlines the most relevant parts of the RISC-V IOMMU specification.
- Chapter 4 delves into the design of the IOMMU IP, identifying the optional features and parameters included and breaking down the microarchitecture of the IP to describe its internal structures.
- Chapter 5 addresses relevant insights concerning the implementation of the IOMMU IP and outlines the development of the RISC-V IOMMU driver for Bao hypervisor.
- Chapter 6 describes the integration of the IOMMU IP into the target platform and covers the validation and evaluation phases. We focus on *Field Programmable Gate Array* (FPGA) resource utilization and translation latencies of DMA transfers, providing quantitative results on each domain for different setups.
- Chapter 7 concludes the document by summarizing all the completed work and the achieved results. In this chapter, we also identify areas for future work related to the IOMMU IP, along with a discussion of alternative designs and architectural/microarchitectural changes to improve further the developed IP and the underlying RISC-V IOMMU specification.

Chapter 2

Background and State of the Art

In this chapter, we present the fundamental background knowledge required in the context of this thesis. First, we start by introducing some basic concepts regarding computer architecture in Section 2.1. Next, we introduce the RISC-V ISA, which has recently gained popularity due to reasons explained further in Section 2.2. Afterward, we present some essential topics related to virtual memory and virtualization, converging towards I/O virtualization and DMA transactions. Subsequently, in Section 2.6, we summarize all related work developed in the field of I/O virtualization and memory protection against DMA-capable devices, with a focus on two main streams: (i) custom state-of-the-art solutions developed in academia and (ii) proprietary IOMMU technologies developed by leading semiconductor design companies. Lastly, we close this chapter with a gap analysis of existing and ongoing efforts in the context of the RISC-V IOMMU.

2.1 The Basics of Computer Architecture

In civil engineering, architects produce plans and models as a form of abstraction for a more comprehensible view of the structures they intend to build. In computer engineering, we need the same type of abstraction to represent a computer. The *architecture* is the programmer's view of the computer. It is defined by the instruction set (language) and operand locations (registers and memory) [14]. The ISA is an abstract interface between the hardware and the lowest-level software encompassing all the information necessary to write a machine language program that will run correctly, including instructions, registers, memory access, I/O, and so on [15]. It does not define the underlying hardware implementation. In effect, a single architecture has multiple hardware implementations. An example is the x86 architecture, mainly used by Intel and AMD to develop different models of general-purpose processors, including those found in most personal computers and servers. These processors can all run the same programs, but they use different underlying hardware. The specific arrangement of registers, memories, *Arithmetic and*

Logic Units (ALUs), and other digital building blocks to form a processor is called *microarchitecture*.

There are two types of architectures: (i) *Complex Instruction Set Computer* (CISC) and (ii) *Reduced Instruction Set Computer* (RISC). In RISC processors, the number of instructions is kept small so that the hardware required to decode the instruction and its operands can be simple, small, and fast. More complex operations that are less common are performed using sequences of multiple simple instructions. Architectures with many complex instructions, such as Intel's and AMD's x86 architecture, are CISC. This type of architecture adds hardware and creates additional overhead that slows down even the simplest instructions. CISC-type architectures can perform a complex task with few or a single instruction executed in several clock cycles. In contrast, RISC-type architectures complete the same task with several instructions, each of which can be executed in few clock cycles. Therefore, the same task is executed with fewer instructions in a CISC processor at the expense of more power consumption and hardware area.

Regarding the source and destination of the data the processor uses, ISAs are classified as *load-store* (also named *register-register*) or *register-memory* architectures. Load-store architectures are commonly used by RISC designs, such as RISC-V, and divide instructions into two categories: (i) memory accesses, which perform data exchange from memory to registers or vice-versa, and (ii) ALU operations, which only occur between registers (e.g., the sum of registers or logic operations between registers). In contrast, register-memory architectures allow ALU operations to be performed on (or from) memory, as well as registers [16].

2.2 RISC-V

RISC-V [10] is an open standard ISA conceived at the University of California, Berkeley, maintained by a non-profit foundation named *RISC-V International*. It is royalty-free and open for anyone to develop proprietary or open-source implementations. Over the last few years, RISC-V has emerged as an increasingly important and disruptive architecture, both pedagogically and commercially [14], due to several differences from other architectures established several years ago. One of the main differences is that RISC-V is a *modular ISA*. The conventional approach adopted by most processor manufacturers is *incremental ISAs*, where new processors must implement not only new ISA extensions but also all extensions of the past (backwards compatibility). This requirement, combined with the marketing appeal of announcing new instructions with a new generation of processors, has led to ISAs that grow substantially in size with age.

RISC-V, instead, is based on RISC principles and follows a modular and extensible approach, where

each core has a base ISA with crucial instructions, which can be either RV32I, RV64I or RV128I¹ depending on the register bit width. These instruction sets are frozen and will never change, giving a stable target to software developers. The base ISA specifies instructions and their encoding, control flow, registers and sizes, memory and addressing, logic (i.e., integer) manipulation, and ancillaries. A simplified general-purpose computer with full software support can be implemented with the base ISA alone.

Name	Description	Version	Status
Base			
RVWMO	Weak Memory Ordering	2.0	Ratified
RV32I	Base Integer Instruction Set, 32-bit	2.1	Ratified
RV32E	Base Integer Instruction Set (Embedded), 32-bit, 16 registers	2.0	Ratified
RV64I	Base Integer Instruction Set, 64-bit	2.1	Ratified
RV64E	Base Integer Instruction Set (embedded), 64-bit	2.0	Ratified
RV128I	Base Integer Instruction Set, 128-bit	1.7	Open
Extension			
M	Standard Extension for Integer Multiplication and Division	2.0	Ratified
A	Standard Extension for Atomic Instructions	2.1	Ratified
F	Standard Extension for Single-Precision Floating-Point	2.2	Ratified
D	Standard Extension for Double-Precision Floating-Point	2.2	Ratified
Zicsr	Control and Status Register Instructions	2.0	Ratified
Zifencei	Instruction-Fetch Fence	2.0	Ratified
Q	Standard Extension for Quad-Precision Floating-Point	2.2	Ratified
L	Standard Extension for Decimal Floating-Point	0.0	Open
C	Standard Extension for Compressed Instructions	2.0	Ratified
B	Standard Extension for Bit Manipulation	1.0	Ratified
J	Standard Extension for Dynamically Translated Languages	0.0	Open
T	Standard Extension for Transactional Memory	0.0	Open
P	Standard Extension for Packed-SIMD Instructions	0.9.10	Open
V	Standard Extension for Vector Operations	1.0	Ratified
Zk	Standard Extension for Scalar Cryptography	1.0.1	Ratified
Hypervisor	Standard Hypervisor Extension	1.0	Ratified
S	Standard Extension for Supervisor-level Instructions	1.12	Ratified

Table 2.1: RISC-V base ISAs and standard extensions. The base ISA and standard extensions highlighted are considered necessary for general-purpose computation and thus have the shorthand "G."

Furthermore, the standard defines optional extensions that hardware can include, depending on the target application, ranging from deeply embedded systems to high-performance computers (e.g., cloud servers and data centers). These extensions are designed to be compatible with each other and with all

¹As of this writing, RV128 is still in open status and has not yet been ratified by RISC-V International

standard bases. Table 2.1 summarizes all RISC-V standard extensions, including those still not ratified (i.e., currently in open or frozen status).

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	<i>Reserved for future use</i>	-
3	11	Machine	M

Table 2.2: RISC-V privilege modes. Adapted from [17].

The RISC-V privileged ISA specification [17] defines three levels of protection or execution modes for RISC-V *harts* (hardware threads), as shown in Table 2.2. Privilege levels are used to provide protection between different components of the software stack. Any attempt to perform operations not permitted by the current privilege mode will cause an exception to be raised. These exceptions will typically cause traps into an underlying execution environment. The Machine-mode (M-mode) has the highest privileges and is the only mandatory privilege level for a RISC-V hardware platform. Code running in M-mode is usually inherently trusted, as it has low-level access to the machine implementation. Many RISC-V implementations will also support at least User-mode (U-mode) to protect the rest of the system from application code, and the Supervisor-mode (S-mode) provides isolation and portability for systems running Unix-like OSes.

2.2.1 RISC-V Cores

Because RISC-V constitutes an open-source ecosystem, there are public RISC-V simulators, compilers, debuggers, and even hardware implementations available written in *Hardware Description Languages* (HDLs) [15]. Recently, many open-source cores have been developed based on the RISC-V ISA. The following subsections describe two RISC-V cores: (i) The CVA6, which the OpenHW Group currently maintains, and (ii) the Rocket Chip, initially developed by the University of California, Berkeley.

a) CVA6 Core

The CVA6 (previously named *Ariane* [18]) is an application-class processor featuring a 6-stage pipeline, single issue, and out-of-order execution stage that implements both 32-bit and 64-bit RISC-V instruction sets, including I, M, A and C extensions as specified in [19]. To enable complete support for Unix-like OSes,

it also implements a *Memory Management Unit* (MMU) for address translation and three privilege levels: Machine (M), Supervisor (S), and User (U). It has optional support for hardware virtualization at design time through the RISC-V Hypervisor extension [5] (see Section 2.4.2). The MMU implements the RISC-V SV39 and SV32 address translation schemes encompassing (i) separate data and instruction *Translation Look-aside Buffers* (TLBs) and (ii) a hardware *Page Table Walker* (PTW) to transform virtual addresses into physical addresses. The CVA6 also has branch prediction units (e.g., *Branch Target Buffer*, *Branch History Table*, and *Return Address Stack*) to deal with control hazards more efficiently.

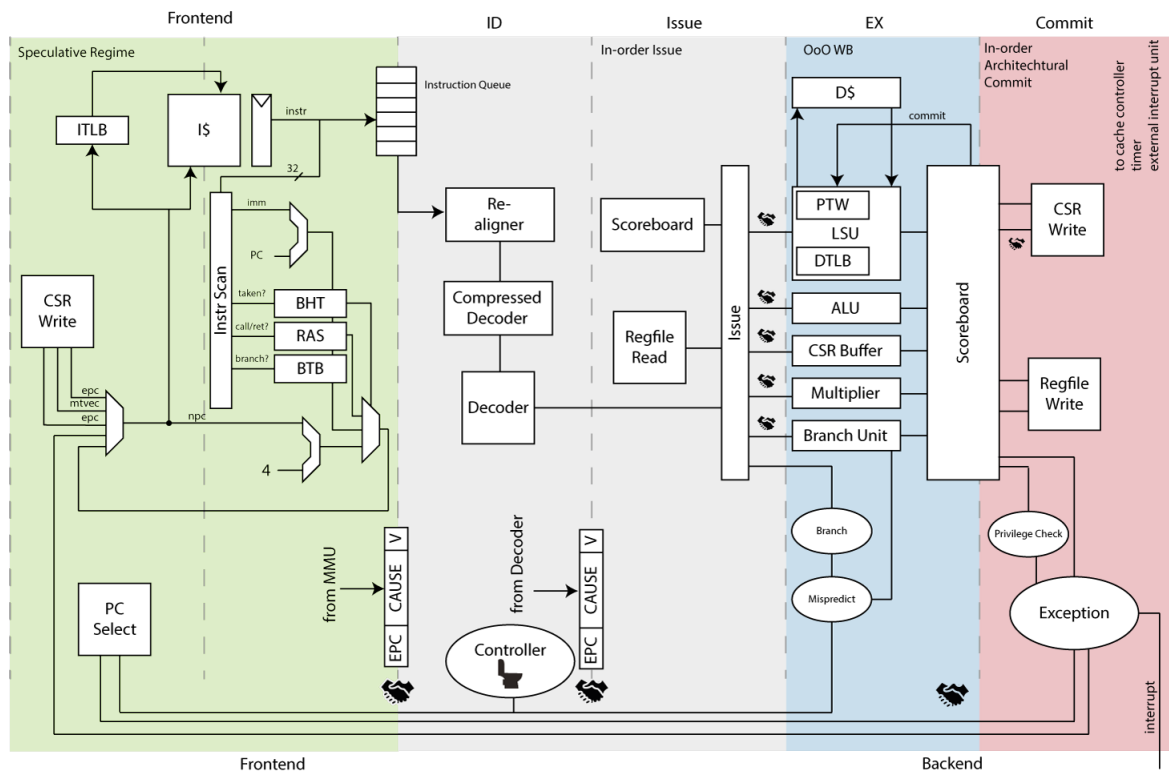


Figure 2.1: CVA6 core microarchitecture (adapted from [18]). The processor's *front end* (green background) encompasses the *program counter generation* and *instruction fetch* stages. The *back end* comprises the *execute* and *commit* stages (blue and red background, respectively). *Instruction decode* and *issue* stages (gray background) perform decoupling through the instruction fetch queue.

Figure 2.1 illustrates an overview of the CVA6 microarchitecture. The program counter generation stage and the instruction fetch stage comprise the processor's *front end*. The front end is fully decoupled from the *back end*, which is in charge of executing the instruction stream. The decoupling is implemented as a *First In First Out* (FIFO) buffer of configurable depth, where instructions are stored in compressed form in the queue, minimizing the number of flip-flops needed. These instructions are re-aligned, decoded, and registered in the *scoreboard* to be further executed in the back end. The execution can be performed

out-of-order to optimize the usage of functional units. Results are placed in the *re-order buffer*, through which write-back conflicts are resolved, and all instructions are committed in program order.

b) Rocket Chip

In the context of RISC-V, the term *Rocket* is associated with two possible alternatives. *Rocket* is a 5-stage in-order scalar processor core generator, initially developed at the University of California, Berkeley [20], and currently supported by SiFive. The generator supports the RV64GC RISC-V ISA and is written in the *Chisel* HDL. It includes page-based virtual memory, a non-blocking data cache, and a front end with branch prediction. Additionally, Rocket cores support the RISC-V Machine, Supervisor, and User privilege execution modes.

The other definition associated to the term *Rocket* is the Rocket Chip SoC Generator, which is an open-source SoC design generator that emits synthesizable *Register-Transfer Level* (RTL) code to be used in FPGAs and *Application-Specific Integrated Circuits* (ASICs). It also leverages the *Chisel* HDL to compose a library of sophisticated generators for cores, caches and interconnects into an integrated SoC. The Chip Generator uses the Rocket Core as a component, which, in combination with L1 data and instruction caches, forms a *Rocket Tile* (i.e., the replicable component of the Rocket Chip SoC generator).

2.3 Virtual Memory

A processor has mainly three functions: (i) reading data, (ii) processing data, and (iii) storing data. During program execution, the processor performs multiple read/write transactions involving the main memory or an external device. Thus, read and write times have a significant impact on a computer system's performance. To optimize memory management, modern computers implement the concept of **memory hierarchy** (see Figure 2.2). With this approach, computer systems are equipped with a few megabytes of high-speed, expensive, volatile² *cache memory*; a few gigabytes of medium-speed, medium-priced, volatile *main memory*; and a few terabytes of slow, cheap, nonvolatile magnetic *Hard-Disk Drive* (HDD) or *Solid-State Drive* (SSD) storage. The responsibility of efficiently managing this memory structure relies on the OS [21], which must keep track of which parts of memory are in use, allocate memory to processes when needed, and deallocate it when they are done.

While early computers did not implement any memory abstraction (i.e., all programs referenced memory through physical addresses), modern computers use a clever technique known as *Virtual Memory*,

²A volatile memory loses its contents when the electric power is switched off.

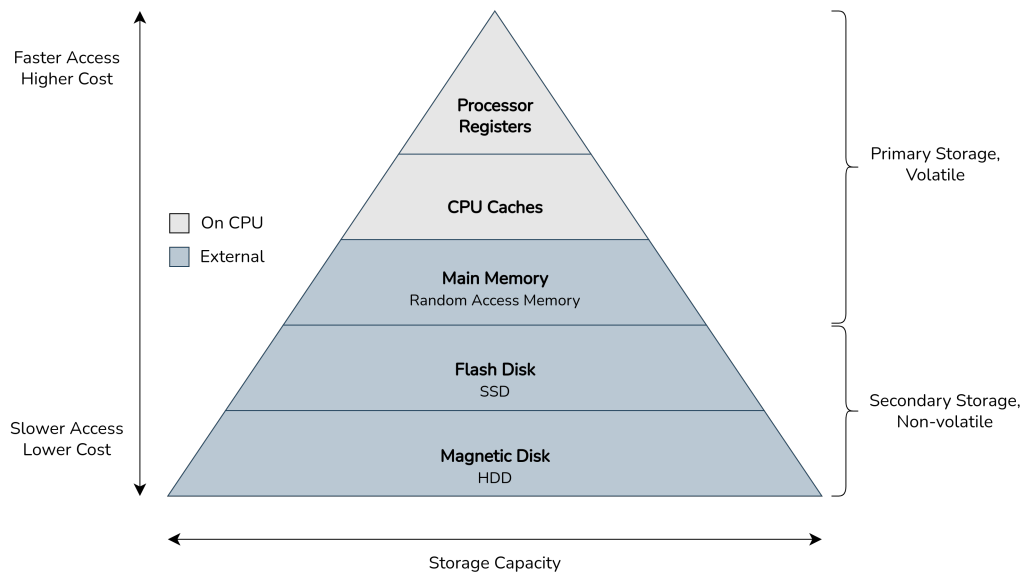


Figure 2.2: Illustration of the memory hierarchy concept: The closer the memory is to the CPU, the higher its price and speed, and the lower its capacity.

where each program has its range of memory locations accessible only by itself. Each of these sets of memory addresses is defined as *address space*, and every address space is divided into fixed-length, contiguous blocks known as *pages*. Every page is mapped onto physical memory, but not all pages have to be physically in the main memory simultaneously. Leveraging the principle of spatial and temporal locality, and the implementation of memory hierarchy, faster translation times can be achieved by maintaining only the pages most used by a program (i.e., the *working set* of the process) in main memory.

When the virtual memory scheme is implemented, all addresses visible by a process are *virtual addresses*, and together, they form a *Virtual Address Space* (VAS). An MMU must first map each of these virtual addresses to a *physical address*, and this can then be used to reference the real physical page in main memory (*page frame*). The MMU is the hardware component responsible for performing this virtual-to-physical translation while enforcing memory protection. The OS assigns a VAS to each process and manages the data structures used to carry out the translation. These structures are referred to as *Page Tables* (PTs). They store all virtual-to-physical page mappings (plus some additional data) and are composed of multiple entries (*Page Table Entries* (PTEs)). Every time a program references a part of its address space that is not present in main memory (an event known as a *page fault*), the OS triggers a fault/exception, gets the missing page in secondary storage, brings it to main memory and re-executes the instruction that failed. A page fault has a significant impact on performance when compared to the opposite case (*page hit*), where the page referenced by software is present in the main memory, and the translation is performed quickly.

A *cache* is an additional level in the memory hierarchy between the processor and main memory. It is used to reduce the number of times the processor needs to access the main memory, which is slower. By storing frequently accessed data in the cache, the processor can access it more quickly, improving the overall performance of the system. In addition, a hardware structure known as a *Translation Look-aside Buffer* (TLB) is integrated into the MMU. It holds the corresponding virtual-to-physical mappings, allowing for faster access times compared to accessing PTs directly. The TLB has a limited number of entries due to its high cost in area and power. A cache can be indexed using physical addresses, virtual addresses, or both. Depending on the indexing method, a cache can be classified as *Physically-Indexed-Physically-Tagged* (PIPT), *Virtually-Indexed-Virtually-Tagged* (VIVT) and *Virtually-Indexed-Physically-Tagged* (VIPT). When physical addresses are needed to index the cache, the TLB is used in conjunction with it to perform the address translation.

2.4 Virtualization

Virtualization is the de facto technology to enable multiple OSES to co-exist in the same hardware platform [22]. In the embedded systems area, it has gained increasing interest due to its advantages in terms of security, reliability, and temporal and spatial isolation [23]. Virtualization is implemented through the *Virtual Machine Monitor* (VMM) or *hypervisor*. This software component is intended to control the execution of multiple VMs by managing system resources. How the hypervisor allows the execution of the different guest OSES depends on the virtualization technique used. Over time, several techniques have been developed to implement the virtualization concept.

The most basic form of virtualization is implemented through **emulation**, where the virtualizing software mimics the hardware provided to the guest OS in the VM. Emulation provides high VM portability and a wide range of hardware compatibility as the guest OS interacts only with the emulated hardware, remaining completely unaware of the virtualization. Typically, emulation is implemented using *interpretation*. The emulator component takes every user mode instruction and translates it to an equivalent instruction suitable according to the underlying hardware. The downside of this scheme is its high cost in terms of performance, which makes it unsuitable for time-constrained systems. In order to obtain better performance, hypervisors try to execute most of the user code directly in the host hardware. However, CPUs featuring different privilege modes have a set of instructions that behave differently when executed in kernel mode than in user mode. There is also a set of instructions that cause a trap if executed in user mode. Popek and Goldberg [24] called these *sensitive* and *privileged* instructions, respectively. Two

techniques emerged to handle virtualization, taking into account these types of instruction:

- **Trap-and-emulate** is a technique that takes the basics of the emulation but improves performance by using interpretation selectively. In this method, both the user applications and guest OS of VMs run in user mode, and the hypervisor runs in privileged mode. Nevertheless, the application runs the instructions natively on the hardware, and only when a privileged instruction is to be executed in virtual user mode a trap to the VMM occurs. The issue prevailing in the trap-and-emulate methodology is that not all ISAs can be emulated using this technique. Those that do not follow the Popek-Goldberg theorem³ cannot be emulated.
- **Binary translation** arose to solve the trap-and-emulate problem by dynamically replacing code with problematic instructions with alternative code sequences that emulated the original instruction [25]. Thereby, sensitive but not privileged instructions are replaced by a trap that, after a sanity check, performs an equivalent instruction and returns the result, whereas other instructions are executed natively.

These two techniques are the basis for software-assisted **full virtualization**, through which guest OSes run on top of the hypervisor without any modification, and the presence of the hypervisor beneath is unknown to the guests. A different form of virtualization is known as **paravirtualization**. It is quite different from full virtualization since it presents a machine-like software interface known as *hypercalls* that explicitly exposes the guest OSes that it is running in a virtualized environment. In this case, guests' source code must be modified to use hypercalls for privileged sensitive operations. Finally, with the advent of CPU extensions for **hardware-assisted virtualization** (e.g., Arm Virtualization Extensions, Intel VT-x, etc), there remains no need to use either binary translation or paravirtualization, as these extensions provide special CPU instructions for privileged and sensitive instructions to trap automatically to the hypervisor, which lies on a new privileged level.

2.4.1 Hypervisor Architectures

According to their hierarchical positioning within the system stack, there are two types of hypervisor architectures [26]:

³Popek and Goldberg stated that a machine is virtualizable only if the sensitive instructions are a subset of the privileged instructions [24].

- **Type-1 or bare-metal hypervisors** (Figure 2.3a) run directly on the host's hardware to control all system resources and manage guest OSes. Examples of open-source bare-metal hypervisors are *Xen* [27], *Bao* [13], and *XVisor* [28].
- **Type-2 or hosted hypervisors** (Figure 2.3b) run as a software layer on top of a conventional host OS. They are used to abstract guest OSes from the host OS. Two open-source hosted hypervisors are *QEMU* and *VirtualBox*⁴.

There is another type of hypervisor known as **Type-3** or **Type-1.5**. These hypervisors combine the power of a full operating system provided by Type-2 and the performance of Type-1. The most common example is KVM. The KVM kernel module turns the Linux kernel into a bare-metal hypervisor (L0). At the same time, the overall system could be categorized as type-2 because the host OS (L1) is still fully functional, and the other VMs (L2) are standard Linux processes from its perspective (see Figure 2.3c).

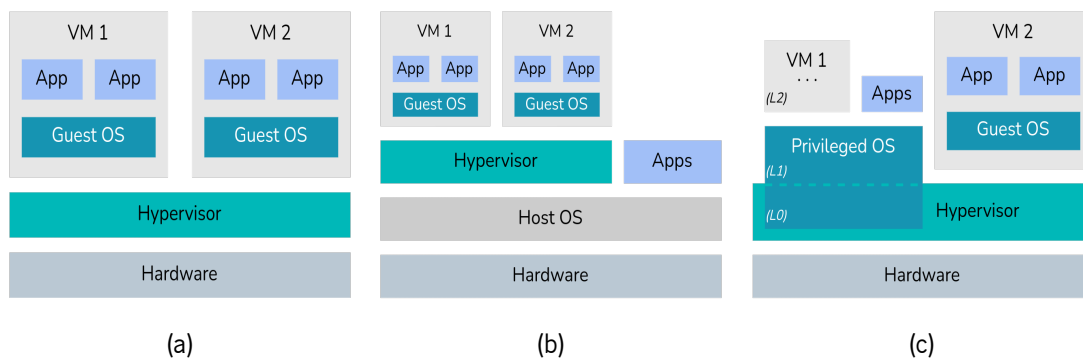


Figure 2.3: Type-1 (a); type-2 (b); and type-1.5 (c) hypervisors. Note the three levels referred for type-1.5 hypervisors.

Regarding the internal structure of the hypervisor, there are two main architectures [29]:

- **Microkernel architectures** are designed to minimize code in the hypervisor space, consisting of a relatively small hypervisor and a separate management partition. In this design, the hypervisor contains only the kernel, while the associated management partition handles networking, storage, hypercalls, and other support functions through an independent OS.
- **Monolithic hypervisors** are completely contained within a single entity. Features such as hardware drivers, application program interfaces, and the virtualization stack are integrated into a thin layer of software that supports virtualization. This platform exists entirely within the most privileged protection ring.

⁴The Extension Package for VirtualBox is not open-source

a) Bao Hypervisor

Bao [13] is an open-source static partitioning hypervisor⁵ that provides strong isolation and real-time and safety guarantees. It implements the pure static partitioning architecture, i.e., a minimal, thin layer of privileged software that leverages the existing ISA virtualization primitives to partition the hardware. Bao has no scheduler and does not rely on any external libraries or privileged VM (e.g., Linux), consisting of a standalone component that depends only on standard firmware to initialize the system and perform platform-specific tasks, such as power management. Bao initially targeted Armv8-A and currently includes support for RISC-V [30], Armv7-A, and Armv8-R.

Regarding I/O, Bao employs a pass-through-only configuration, directly assigning peripherals to guests. The hypervisor does not verify the exclusive assignment of a given peripheral, which allows for several guests to share it, albeit in a non-supervised manner. Hence, integrating RISC-V IOMMU support in Bao would enhance system-level protection for platforms implementing virtualization using this hypervisor.

2.4.2 RISC-V Hypervisor Extension

Virtualization mode	Nominal Privilege	Name	Abbreviation
0	U	User mode	U-mode
0	S	Hypervisor-extended supervisor mode	HS-mode
0	M	Machine mode	M-mode
1	U	Virtual user mode	VU-mode
1	S	Virtual supervisor mode	VS-mode

Table 2.3: RISC-V privilege modes with the hypervisor extension (adapted from [17]). The virtualization mode indicates whether the RISC-V hart is executing in a guest.

The RISC-V ISA was designed from the ground up to be classically virtualizable by allowing to selectively trap accesses to virtual memory management *Control and Status Registers* (CSRs) as well as timeout and mode change instructions from supervisor/user to machine mode. Furthermore, RISC-V provides precise exception handling, guaranteeing the ability to fully specify the instruction stream state at the time of an exception. Therefore, the simplicity of the ISA, coupled with its virtualization-friendly design, allows the seamless execution of a hypervisor recurring to traditional techniques. Nonetheless, the RISC-V

⁵A static-partitioning hypervisor separates the hardware (CPU cores, physical memory and I/O devices) of a physical machine into different guest domains that directly manage their assigned resources.

privileged architecture was endowed with hardware virtualization support to assist hypervisors and reduce virtualization overhead while enhancing efficiency through optional extensions.

In a non-virtualized environment, the RISC-V privileged specification [17] defines three execution modes: (i) *Machine Mode*, (ii) *Supervisor Mode*, and (iii) *User Mode*. When the hypervisor extension is implemented, it extends these levels to those outlined in Table 2.3. Hence, the supervisor mode is extended to the *Hypervisor-extended Supervisor* mode. This execution mode is orthogonal to the new *Virtual Supervisor* and *Virtual User* modes to fit type-1, type-2, and hybrid hypervisors.

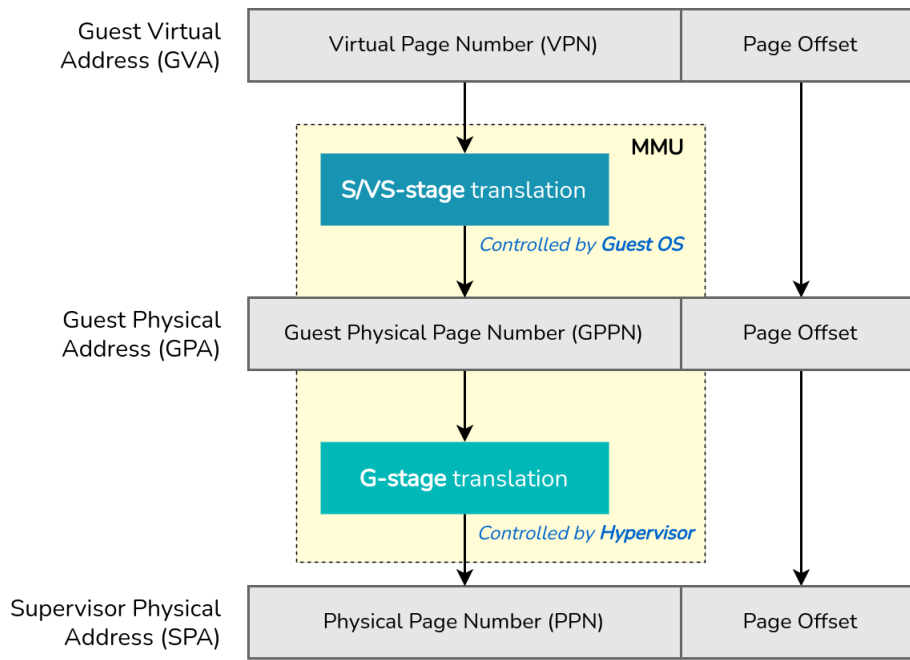


Figure 2.4: RISC-V two-stage address translation flow.

At the virtual memory level, the RISC-V hypervisor extension introduces two-stage address translation as depicted in Figure 2.4: (i) *S/VS-stage* to translate *Guest Virtual Addresses* (GVAs) to *Guest Physical Addresses* (GPAs) using the *S/VS-stage* PTs hold in main memory and controlled by the corresponding guest OS, and (ii) *G-stage* to translate GPAs to *Supervisor Physical Addresses* (SPAs) using the *G-stage* PTs hold in main memory and controlled by the hypervisor. Four translation schemes are defined by the privileged ISA specification [17] for each stage to define some properties of the translation process (e.g., address size and PTE formats). The translation schemes defined for the *S/VS-stage* are Sv32, Sv39, Sv48, and Sv57. Then, the hypervisor extension extends these schemes to Sv32x4, Sv39x4, Sv48x4, and sv57x4 for *G-stage*.

2.4.3 IO Virtualization

When running in a virtualized environment, the VMM must virtualize I/O devices to the guest OS. Since instructions to access I/O devices change the configuration of the system (i.e., are sensitive), each attempt to access an I/O device will trap to the hypervisor in architectures meeting the Popek-Goldberg theorem, generating a significant overhead. In addition, multiple guest OSes can access the same I/O device, each believing it has absolute control over the device, which may lead to access conflicts and data corruption. Moreover, there is the DMA problem mentioned in Chapter 1, where DMA-capable devices use absolute physical addresses to reference main memory and can access arbitrary memory regions. These issues fuel an area of study within virtualization dedicated to managing I/O devices.

According to Intel [7], there are four I/O virtualization techniques that a hypervisor can use:

- **I/O Emulation**

It is also known as *Full Virtualization*. The hypervisor exposes a virtual device to guest software by emulating an existing *legacy* I/O device over whatever physical devices are available on the physical platform. The hypervisor traps I/O-related operations and emulates them.

- **I/O Paravirtualization**

Similar to I/O emulation, but instead of emulating legacy devices, the VMM exposes a software interface to the guest. This model provides improved performance over emulation at the cost of reduced compatibility and increased engineering efforts due to the need for specialized guest software.

- **Device Pass-through**

The hypervisor may directly assign physical I/O devices to VMs. In this model, the driver for an assigned I/O device runs in the VM to which it is assigned and is allowed to interact directly with the device hardware with minimal or no VMM involvement.

- **I/O Device Sharing**

In this model, which is an extension to the device assignment model, an I/O device supports multiple functional interfaces, each of which may be independently assigned to a VM. The device hardware itself is capable of accepting multiple I/O requests through any of these functional interfaces and processing them utilizing the device's hardware resources.

An IOMMU enables a flexible management of I/O devices within a virtualized environment. In the device assignment scheme, the IOMMU allows the addresses to be remapped transparently, and both the device and the VM are blissfully unaware of the address translation under the hood [21]. Furthermore, it ensures that a device assigned to a VM can directly access the memory associated with that VM without jeopardizing the integrity of the other guests. Direct access means that the IOMMU controls device accesses to memory with none/minimal VMM intervention, so trap-and-emulate is no longer necessary, reducing the number of VM exits and entries and therefore increasing system performance. In the case of static-partitioning hypervisors like *Bao* [13], these features are crucial as system resources are directly assigned to VMs at boot time, and devices are not emulated. Going beyond DMA and address translations, an IOMMU also virtualizes interrupts so that the interrupt generated by a device arrives at the right VM with the correct interrupt number (*interrupt remapping*).

2.5 Direct Memory Access

There are two traditional ways of performing I/O transactions [21]: (i) *Programmed I/O*, which relies on transitions between user space and kernel space through system calls to dispose of system hardware resources using *polling-based* techniques, and (ii) *Interrupt-Driven I/O*, which allows the CPU to continue execution while waiting for the I/O resources to become ready on every atomic transaction, moment at which an interrupt is generated and the control is taken by an *Interrupt Service Routine* (ISR) to continue the I/O operation. A disadvantage of these schemes is the need for more CPU time. Polling-based techniques tie up the processor full-time until the whole I/O transaction is done, while interrupt-driven I/O generates an interrupt on every transaction. Instead, a mechanism known as *Direct Memory Access* (DMA) allows peripheral devices direct access to memory without CPU intervention [31]. With DMA, the CPU programs the DMA controller registers with the memory transfer parameters. The CPU is then free to perform unrelated work while the DMA engine performs the copy. Once the copy operation is complete, the DMA engine sends an interrupt to the CPU to indicate that the memory transfer has been completed.

2.5.1 DMA as a Potential Threat

Since DMA engines can access an arbitrary range of memory physical addresses, misbehaving I/O controllers can use DMA capabilities to tamper with critical memory data (e.g., OS kernel code or data). Such malicious actions are referred to as DMA-based attacks and can be classified into two categories [32]:

1. The first category concerns I/O controllers that need the cooperation of the processor to carry out memory accesses. As these controllers cannot independently access system memory, it is necessary to make the targeted OS execute some software to tamper with the memory using DMA.
2. The second category concerns I/O controllers that can autonomously access the main memory. These controllers must own a DMA engine and must be able to master the I/O bus they are connected to. Such I/O controllers are called DMA bus-mastering controllers (or, simply, bus masters) [33].

Having an IOMMU in place ensures that every memory access issued by DMA-capable peripherals is subjected to address translation and permission checks. Thus, the IOMMU behaves like a firewall for DMA engines, filtering out unauthorized accesses to main memory. From the security point of view, these features allow to cope with the most significant part of DMA attacks immediately.

2.6 Related Work

For the past years, there has been a lot of research and development in the field of I/O virtualization. Academia has actively participated by creating customized solutions for different application fields. Meanwhile, mainstream semiconductor design companies have developed specialized hardware not only to introduce virtualization support for CPU and memory but also for I/O devices. For instance, Intel and AMD introduced in the market their first processors with support for hardware-assisted virtualization [34]. In this section, we summarize some advancements that have been made in the development of I/O virtualization technology. The structure of this section is as follows: We start by describing some custom works related to this area that have been developed in academia, focusing on those targeting embedded systems. Then, we summarize relevant features of proprietary IOMMUs developed by some leading semiconductor companies (e.g., Intel Vt-d, AMD-Vi, and ARM SMMU). Furthermore, we present an overview of the RISC-V IOMMU and conduct a comparative analysis about these proprietary solutions. Lastly, we perform a gap analysis of existing IOMMUs developed in RISC-V platforms.

2.6.1 Custom IOMMU-related Solutions

Academia has researched and proposed different approaches to enable efficient and secure management of I/O devices in embedded systems. Some of these approaches are entirely software-driven ([35], [36], [37], [38], [39]), relying on custom scheduling schemes or the introduction of additional abstraction

layers via virtualization. This is done to avoid the need for incorporating extra hardware, which can increase system costs and complexity. In contrast, others opt for the integration of sophisticated hardware modules into hybrid systems ([40], [41], [42], [43]), arguing that this approach can yield better results when compared to exclusively software-based solutions.

a) Software-based

Regarding software-based solutions, Golchin et al. [35] developed *Boomerang*, an I/O system comprising real-time task pipelines in a static-partitioning hypervisor, which connects the Quest *Real-Time Operating System* (RTOS) with a legacy non-real-time OS such as Linux, via secure and predictable shared memory communication channels. This system was designed so that the legacy OS benefits from timing predictable services that are isolated from less critical code, and, at the same time, the RTOS benefits from pre-existing services of the legacy non-real-time system. Experiments showed that real-time task pipelines guarantee end-to-end throughput, delay, and loss requirements.

While *Boomerang* operates at the hypervisor level, West et al. [36] [37] took a different approach by modifying the internal workings of the Quest RTOS, also utilized in *Boomerang*. They developed a scheduling technique focusing on predictability, encompassing the integrated management of tasks and I/O events, including interrupts. Furthermore, the scheduler was integrated into an *adaptive mixed-criticality* framework embedded within the Quest kernel. This extension empowered Quest with support mode changes between various criticality levels, allowing components of different levels to coexist within a single Quest-V sandbox or a unified Quest system.

Another scheduling-based approach was investigated by Masrur et al. [38]. They optimized the worst-case behavior of the standard *Simple Earliest Deadline First* scheduler of the Xen hypervisor [27] through appropriate modifications. Additionally, they proposed a new custom scheduler called *Priority-based Simple Earliest Deadline First*, which is based on the concept of a real-time VM. This VM is scheduled separately using a fixed-priority scheme, reducing latency and jitter.

Alternatively, Schwarz and Gehrmann [39] employed virtualization to protect memory in systems based on ARMv5 and newer ARM architectures. This was achieved by emulating a DMA controller for each guest mode using the standard MMU. Such an approach allowed the hypervisor to manage DMA tasks, ensuring proper handling of access conditions and scheduling issues. After DMA task completion, the hypervisor routed the received interrupt from the corresponding physical DMA controller to the respective guest as needed.

b) Hardware-based

Purely software-based systems offer cost-effectiveness and simplicity by eliminating the need for additional hardware components. However, in many scenarios, the need for dedicated hardware support for I/O device management and virtualization can increase the CPU's workload, potentially preventing a real-time system from meeting crucial requirements like predictability and determinism [44]. Furthermore, many software-based solutions do not address security and system integrity against malicious devices. Therefore, most approaches to these challenges integrate dedicated hardware modules to handle the additional processing load.

Betti et al. [40] developed a real-time I/O management system supporting various priority-based scheduling policies for *Commercial Off-The-Shelf* (COTS) peripherals and interconnect systems. This system efficiently utilizes real-time bus resources without compromising peripheral throughput. It also implements I/O device virtualization techniques to schedule multiple real-time I/O flows within the same peripheral and enable secure device sharing among tasks with different priorities. The system comprises two key components: a *real-time bridge* and a *peripheral scheduler*, both implemented on FPGA. These components were evaluated, demonstrating deterministic delays while meeting all defined I/O deadlines.

Münch et al. [41] introduced the concept of an *IO Memory Protection Unit* (IOMPU) aimed at providing spatial isolation for I/O within a system using COTS components and *non-transparent bridge technology*. The IOMPU intercepts any memory-mapped I/O interconnect, enhancing system security. Additionally, it is compatible with existing temporal isolation solutions and incorporates a built-in safe blocking feature. The evaluation revealed a performance overhead below 0.88%, which is almost negligible, particularly compared to previously referred software-based solutions.

Jiang et al. [42] proposed a novel I/O architecture known as *MCS-IOV*, which leverages hardware-assisted virtualization to offer temporal and spatial isolation, preventing fault propagation with minimal performance impact. MCS-IOV facilitates I/O virtualization and the possibility of switching to an I/O-driven criticality mode, establishing VMs as independent execution environments called *criticality worlds*. It extends the existing real-time I/O system *BlueIO* [45], supporting mixed criticalities and custom scheduler interfaces, thereby enhancing timing predictability. This system was implemented and evaluated on FPGA, demonstrating favorable results regarding memory footprint, hardware resource consumption, I/O throughput, and predictability.

In the automotive area, Sriramakrishnan et al. [43] devised a hybrid system to support deterministic I/O translation. This system employs a hardware look-up-table-based particular path to manage real-time I/O traffic. It offers a mechanism for software-configurable runtime selection of the conventional

IOMMU scheme to ensure flexibility and scalability. The real-time path encompasses two stages: (i) a *Physical Address Translation* module and (ii) a *Peripheral Virtualization Unit* module. Testing with two VMs, one running a critical application and the other running an infotainment application, yielded near-native performance, achieving approximately 98% compared to a configuration with no IOMMU.

2.6.2 Intel VT-d

Intel's *Virtualization Technology for Directed I/O (VT-d)* [7] has been included in most high-end newer Intel processors since the Core 2 architecture as a complement of the VT-x technology for hardware-assisted CPU virtualization. In brief, it assists physical memory virtualization for I/O controllers and enhances memory protection for device accesses. Intel VT-d enables (among others) the following main features: (i) I/O device assignment, (ii) DMA remapping through single-stage or two-stage (nested) address translation, (iii) interrupt remapping, and (iv) interrupt posting.

Intel VT-d encompasses a set of DMA Remapping Hardware Units, which are responsible for remapping all memory accesses originating from I/O controllers and performing memory permission checks. A single remapping unit can manage one or more controllers. Each remapping unit exposes a set of registers placed at a 4KB-aligned memory-mapped location, through which it can be configured. The specification also includes a performance monitoring infrastructure.

a) DMA Remapping Units

Intel VT-d DMA remapping units perform device address translation and permission checks for every memory access. Device domains are abstractly defined as isolated environments in the platform to which a subset of the host physical memory is allocated. Each inbound request appearing at the address-translation hardware must identify the device originating the request. The attribute identifying the initiator of an I/O transaction is denoted *source_id* in the specification. The top-level structure to map devices to their respective domains is the *root table*. Each *root-table* entry contains the context-table pointer, which references the *context-table* for all the devices on the bus identified by this entry. A context-table entry maps a specific I/O device on a bus to the domain to which it is assigned and, in turn, to the address translation structures for the domain.

If nested translation is enabled, all memory accesses generated when processing a request through first-stage translation are subjected to second-stage translation. This includes accesses to first-stage paging structure entries (i.e., PML5E, PML4E, PDPE, PDE, PTE), and accesses to the output address from first-

stage translation, as shown in Figure 2.5. For requests subjected to nested translation, access rights are checked at both the first and second stages of translation. Additionally, the DMA remapping hardware may also be programmed to perform pass-through (i.e., no address translation or memory access rights checks).

Remapping hardware may accelerate the translation process by caching data from the memory-resident paging structures. Intel VT-d defines four types of caches that may be included: (i) Context cache, (ii) *Process Address Space ID (PASID)* cache, (iii) *I/O Translation Look-aside Buffer (IOTLB)*, and (iv) paging-structures caches. Because the hardware does not ensure that the data it caches is always consistent with the structures in memory, the software is responsible for taking actions to invalidate cached data that may be inconsistent. The architecture also supports enabling endpoint devices to participate in the remapping process with translation caches implemented at the device itself (device TLBs).

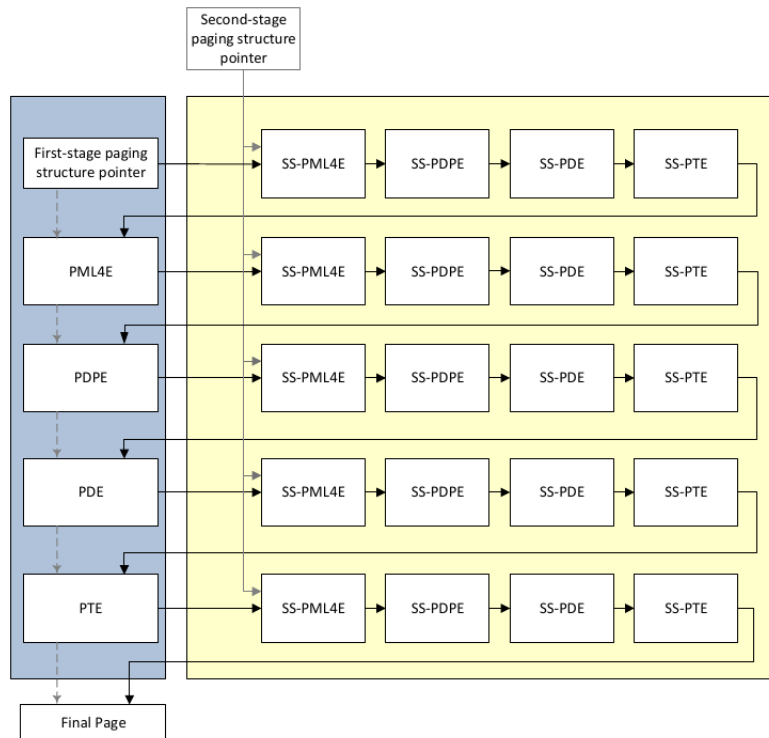


Figure 2.5: Intel VT-d nested translation with 4KB pages. Adapted from [7].

b) Interrupt Remapping and Posting

Intel VT-d's interrupt-remapping architecture enables system software to handle external interrupt requests generated by all sources (e.g., interrupt controllers and *Message-Signaled Interrupt (MSI)*/MSI-X capable devices). Interrupt requests appear to the root-complex as upstream double-word sized memory write requests, and consequently, interrupt-remapping is co-located with the remapping hardware units.

The architecture utilizes a memory-resident single-level table called *Interrupt Remapping Table*. Moreover, the specification defines an additional capability named *interrupt posting*, which enables a remappable format interrupt request to be posted/recorded in a coherent memory-resident data structure, with an optional notification event to the CPU complex to signal pending posted interrupt.

2.6.3 AMD-Vi

AMD I/O Virtualization Technology, AMD-Vi, or AMD IOMMU [8] are the names used to refer to the I/O virtualization technology developed by AMD. It extends the AMD64 system architecture by adding support for address translation and system memory access protection on DMA transfers, as well as filtering and remapping interrupts from peripheral devices. For reasons of simplicity, in the remainder of this section, the term IOMMU will be used to refer to the AMD-Vi technology. Similarly to Intel VT-d, this technology implements the concept of protection domains, using a 16-bit identifier denominated *deviceID*. The *Device Table* holds specific information for each device domain and is indexed using the *DeviceID*.

The AMD-Vi technology supports two-stage address translation to convert guest-physical to host-physical address. Each Device Table entry is extended to include optional address translation information for guest-virtual to guest-physical address translation. Guest address translation supports 5-level or 4-level 4KB page table formats as defined by the AMD64 processor. In contrast, host address translation uses a page table structure designed to support a complete 64-bit device virtual address space while allowing faster translation in many common cases. The data structures for nested translation are very similar to those specified by Intel for 4KB nested translations (see Figure 2.5). AMD-Vi also supports local IOMMU cache design and remote IOTLBs (device TLBs), allowing a trusted I/O device with IOTLB support to cooperate with the IOMMU to maintain its own cache of address translations. In addition, there is a set of event counters and IOTLB commands that can be used to monitor performance and synchronize IOTLB data.

a) Memory-based Queue Interface

An advantage over Intel VT-d technology is the inclusion of three memory-based queues for exchanging command and status information between the IOMMU and the system processor(s):

- Host software communicates with the IOMMU through a shared circular buffer in system memory named *Command Buffer*. The host software writes commands into the buffer and then notifies the IOMMU of their presence by writing a new value to the tail pointer (see Figure 2.6).

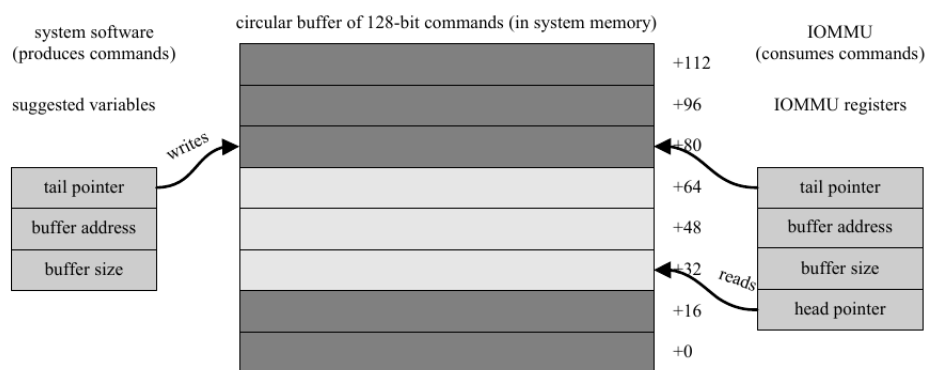


Figure 2.6: AMD-Vi command buffer in system memory. Adapted from [8].

- The IOMMU reports atypical events to the processor using another circular buffer, the *Event Log*, located in system memory.
- *Peripheral Component Interconnect Express (PCIe) Address Translation Services (ATS)* and *Page Request Interface (PRI)* services can be used together to enable the peripheral to use unpinned pages for I/O, allowing a system to operate efficiently under low available memory. Peripheral page requests in AMD-Vi are reported to the host software through a shared circular buffer in system memory: the *Peripheral Page Request Log*.

b) AMD-Vi Emulation

The IOMMU was designed from the ground up to be classically virtualizable so hypervisors can easily emulate its behavior to a guest OS. To cope with the additional overhead associated with the emulation of IOMMUs, AMD-Vi may optionally support the *vIOMMU* feature, which provides partial hardware acceleration when emulating guest IOMMUs. *vIOMMU* provides acceleration when a VM accesses its Command Buffer, Peripheral Page Request Log, and Event Log. This eliminates the CPU overhead needed for the supporting hypervisor intercepts and reduces the latency of these operations.

c) Interrupt Remapping and Virtualization

AMD-VI provides interrupt remapping tables specified by each Device Table to support permission checking and interrupt remapping for the corresponding I/O device interrupts. Moreover, it optionally supports interrupt direct assignment to guest VMs without hypervisor intervention. This feature requires compatible *Advanced Programmable Interrupt Controller* virtualization support in the AMD processor.

2.6.4 ARM SMMU

ARM's IOMMU is known as the *System Memory Management Unit* [9], and it is currently in version 3 (ARM SMMUv3). Similarly to Intel and AMD, ARM defines an identifier (i.e., *StreamID*) to differentiate each client device interacting with the SMMU. It is used to index a *Stream Table Entry* in a *Stream table*, which contains per-device configuration. When the SMMU implements stage-1 translation, another identifier (i.e., *SubstreamID*) might optionally be provided to identify different application contexts associated with the same device. An implementation encompassing only stage-2 translation does not take a *SubstreamID* input.

A Stream table entry holds the stage-2 translation table base pointer, as well as the location of the stage-1 configuration structures. A *Context Descriptor* represents stage-1 translation, and a Stream table Entry represents stage-2 translation. The Context Descriptor holds the stage-1 translation table base pointer (to translate the input GVA to a GPA), per-stream configuration, and *Address Space ID* (ASID). If stage-2 is used, each Stream Table Entry contains the stage-2 translation table base pointer (to translate GPA to the output SPA), per-device configuration, and *Virtual Machine ID* (VMID). Figure 2.7 illustrates a basic configuration of these data structures.

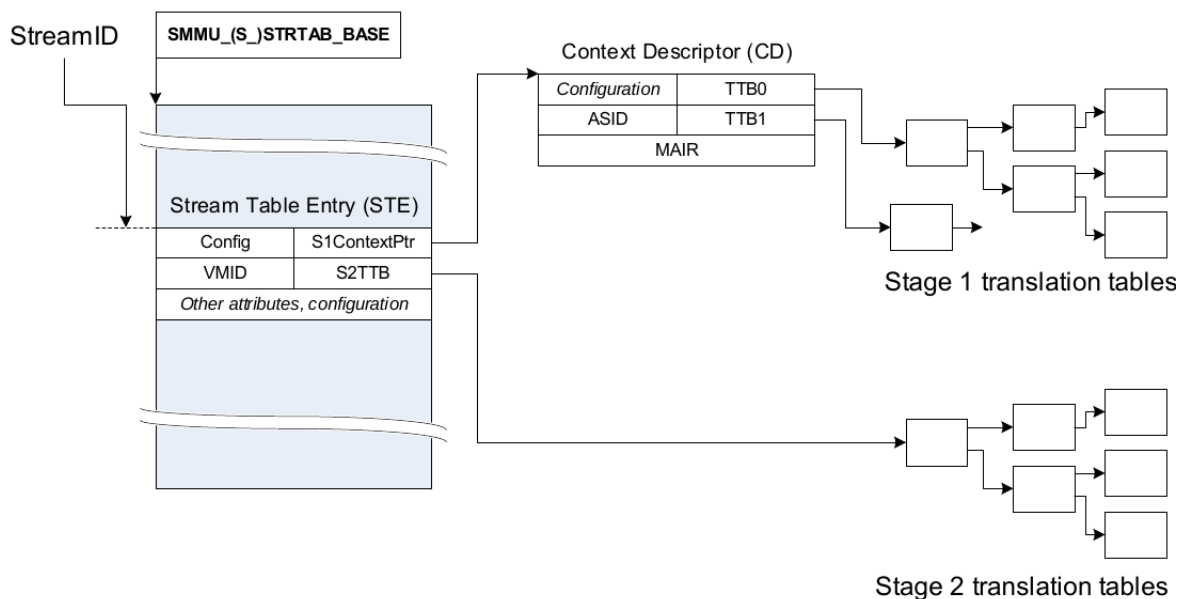


Figure 2.7: ARM SMMU context data structures. Adapted from [9].

The ARM SMMU supports locally caching translation and configuration structures through caches/IOTLBs to improve lookup times and reduce latencies caused by long memory access times. Cached translations are tagged with a translation regime, an ASID, and a VMID to prevent aliasing between different translations. The ARM SMMU also includes optional performance monitoring features that allow for the detection

of specified events related to the SMMU or its implementation and the tracking of these events through counters that increment when the configured event occurs. Additionally, the SMMU specification defines three memory-based circular queues, similar to those found in AMD systems (i.e., *Command queue*, *Event queue*, and *PRI queue*).

a) Security States

A unique feature in the ARM SMMU is that the architecture supports two security states, each with an associated physical memory address space: *Secure state* and *Non-secure state*. When the secure state is implemented, the SMMU includes a unique set of registers only accessible in the secure world to configure the state, a *Secure Command queue*, a *Secure Event queue*, and a *Secure Stream table*. If a stream is identified as secure, its configuration is obtained from the secure Stream table or from secure registers. If it is identified as non-secure, its configuration is obtained from the non-secure Stream table or from non-secure registers. A transaction from a secure stream can result in both secure and non-secure transactions to the memory system. In contrast, a transaction from a non-secure stream can only result in non-secure transactions to the memory system.

b) Interrupt Generation

The SMMU incorporates support for generating interrupts related to Event queue recording, PRI requests, and global errors. To trigger these interrupts, an implementation must support either one of these options, or both: wired interrupts and MSIs. MSI events originating from secure sources are sent using secure accesses, while those from non-secure sources are written through non-secure accesses. It is important to note that the ARM SMMU specification does not provide any information regarding interrupt remapping capabilities.

2.6.5 An Overview of the RISC-V IOMMU

The RISC-V IOMMU uses two memory-based data structures to store context information: the *Device Context* (DC) and the *Process Context* (PC). The DC associates a device with an address space and holds per-device configuration parameters required for address translation. Similarly, the PC associates a process with a virtual address space in devices that support multiple process contexts (e.g., PCIe devices). Additionally, it keeps controls for first-stage translation. Context data serves to locate first and second-stage PTs. The IOMMU uses identical virtual memory schemes, PT formats, and two-stage address translation

process as defined in the RISC-V Privileged Specification [17]. Furthermore, the RISC-V IOMMU optionally support redirecting MSIs from guest-controlled devices to the corresponding interrupt controller. To minimize latencies introduced by walking memory for locating context and mapping data structures, the IOMMU incorporates *IO Address Translation Caches* (IOATCs) to store translation-related data in hardware structures with faster access times. First and second-stage PT data is stored in the IOTLB, while DCs and PCs are cached in the *Device Directory Table Cache* (DDTC) and *Process Directory Table Cache* (PDTC), respectively.

To interact with software, the RISC-V IOMMU integrates three interface subsystems: (i) A memory-mapped register interface; (ii) three memory-based queues (*Command Queue* (CQ), *Fault/Event Queue* (FQ) and *Page-Request Queue* (PQ)); and (iii) an interrupt generation subsystem. The memory-mapped interface is used by software to discover IOMMU capabilities, configure the IOMMU hardware, and monitor internal modules. The CQ allows software to issue commands to the IOMMU hardware for synchronization (e.g., IOATC invalidation and memory barriers). The FQ is the channel to report faults and events arising from the processing of transactions to software, accompanied with associated data, and the PQ is used for PCIe ATS and PRI features. The IOMMU can use two mechanisms to generate interrupts requesting software service: *Wire-Signaled Interrupts* (WSIs) and MSIs. At least one of these mechanisms is obligatorily supported, while including the other remains optional. In addition to these three mechanisms, software can leverage two complementary features of the RISC-V IOMMU: a *Hardware Performance Monitor* (HPM) equipped with event counters for gathering performance-related data and a register-based debug interface for explicitly requesting the IOMMU to translate a user-defined *IO Virtual Address* (IOVA).

2.6.6 IOMMU Comparison and Gap Analysis

The remainder of this chapter performs a comparative analysis between the three proprietary IOMMUs described in the previous sections and the RISC-V IOMMU⁶, which constitutes the central point of this thesis. Additionally, we collect information about the existing and ongoing works related to the RISC-V IOMMU to identify discrepancies and gaps within this domain.

Table 2.4 presents a comparison between the RISC-V IOMMU, ARM's SMMU, AMD-Vi and Intel VT-d in terms of relevant architectural features. Overall, Intel and AMD's IOMMUs mainly aim at high-performance and general-purpose platforms, including features not typically used in real-time or embedded applications while lacking some crucial features for embedded platforms. For example, only Arm and RISC-V specify

⁶Even though Chapter 3 provides a comprehensive description of the RISC-V IOMMU specification, we presented a brief overview in Section 2.6.5 to provide the required context for these comparisons.

Feature	Arm SMMU	AMD-Vi	Intel VT-d	RISC-V IOMMU
Behavior when disabled	Configurable. Block (abort) or bypass	Bypass all bus traffic without alteration	Not specified	Block
Context data structures	Two separated context tables, one for first-stage data, another for second-stage data	Single context table for all context data	Legacy mode: Single context table for second-stage data. Scalable mode: Context table points to PASID tables to save first and second-stage data	Single two-stage context table (no PASID), additional first-stage context table (transactions with PASID)
Software interface	Register interface with multiple 64-kB pages. Command queue, event queue and PRI queue	512-kB register interface. Command buffer, event log and PRI log	Each DMAR has a 4-kB register interface, an invalidation queue and a PRI queue	4-kB register interface. Command queue, fault/event queue and PRI queue
Support for translation caches	Configuration caches and translation cache. Invalidations performed through command queue	Device table cache and translation table cache. Invalidations performed through the command buffer	Context cache, PASID cache, IOTLB, paging-structure caches. Two invalidation mechanisms: register-based and queue-based	First and second-stage context caches, IOTLB. Invalidations through the command queue
Interrupt generation	Wired interrupts and MSI	Only MSI or MSI-X mechanisms. Support for wired interrupt not specified	Only MSI or MSI-X mechanisms. Support for wired interrupt not specified	Wired interrupts and MSI
Interrupt remapping	Not specified	Interrupt remapping and virtualization	Interrupt remapping. Includes support for posting interrupts in a memory-resident data structure	Interrupt remapping and virtualization. Includes support for posting interrupts in a memory-resident data structure
Fault models, recording and reporting	Queue-based fault reporting. Two fault models: Stall and terminate . Supports compression of multiple faults from the same requester	Queue-based fault reporting	Register-based fault reporting. Two types of fault: non-recoverable (aborted) and recoverable . Supports compression of multiple faults from the same requester	Queue-based fault reporting
Support for embedded implementations	Yes	No	No	No
IOMMU virtualization	Hardware support not specified. Suggests VMM hardware emulation or paravirtualization	vIOMMU: Hardware acceleration to implement guest IOMMUs. Also provides VMM hardware emulation guidelines	Hardware support not specified. Suggests VMM hardware emulation or paravirtualization	Hardware support not specified. Suggests VMM hardware emulation or paravirtualization
Support for security states	Two security states (worlds)	Not specified	Not specified	Not specified
Performance monitoring	Implementation-defined number of event counter groups (1-64 counters each). 8 standard events, including clock cycles	1-63 counter banks (2-15 counters each). 25 standard events, including free-run configuration	1-2 ¹⁶ counters. 20 standard events (including clock cycles) grouped into 1-15 event groups	1-31 event counters + free clock cycles counter. 8 Standard events
Debug support	Implementation-defined	Not specified	Not specified	Register-based interface for explicit address translation of user-defined IOVAs

Table 2.4: Architectural comparison between the RISC-V IOMMU and other proprietary IOMMUs developed by Arm, AMD, and Intel.

WSI *Interrupt Generation Support* (IGS), whereas AMD and Intel only include support for MSI and MSI-X, commonly associated with PCIe-compatible platforms. Additionally, AMD introduces partial hardware acceleration to implement guest IOMMUs. The *vIOMMU* feature within AMD-Vi exposes memory-mapped resources for implementing part of each VM's guest IOMMU memory-mapped register space.

In turn, RISC-V and Arm's IOMMUs also address the embedded domain. However, due to its recent introduction, the RISC-V IOMMU lacks some key features specified in the Arm architecture. For example, Arm uses two configurable models to handle translation faults, with an option for grouping multiple faults

from the same requester. In the *terminate* model, faulting transactions are either aborted or completed with reads returning zero and writes being ignored. In the *stall* model, faulting transactions are buffered in a stalled state until subsequent resolution. Moreover, Arm supports embedded SMMU implementations for platforms with limited resources, using on-chip or internal storage for context data and interface queues, eliminating the need for software to allocate memory for these structures.

Regarding security, the Arm SMMU architecture enables shared use of an SMMU between *secure* and *non-secure* execution domains. The first step of transaction processing identifies which of these domains a transaction belongs to and, therefore, the set of resources that process the transaction. Regarding the RISC-V architecture, a similar security model was recently provided by SiFive to RISC-V international and is starting to be adopted by several RISC-V platforms. RISC-V's *WorldGuard* offers fine-grained security domains (*worlds*) for isolated code execution and data protection. It provides platform-level information and isolation control, featuring multiple privilege levels per world and supporting an unlimited number of worlds. WorldGuard ensures multi-domain security, providing data protection for core, cache, interconnect, peripheral, and memory.

a) RISC-V IOMMU Gap Analysis

Given the recent ratification of the RISC-V IOMMU specification, only some implementations currently comply with this standard in the industry. Table 2.5 summarizes the existing and ongoing implementations of the RISC-V IOMMU. This summary provides relevant information about the architectural features of these IPs and the organizations or processor families they are associated with.

IOMMU's Organization/Processor	License	Address Trans.	MSI Trans.	IOATCs	PCIe	IGS	HPM	Parameterizable Design	Processor's Target Application	Status
Rivos	BSD 2-Clause License	Sv39(x4) & Sv48(x4)	✓	DDTC & PDTC	✓	?	✗	✗	Reference use	WiP
SiFive's P600/P400 Series	Commercial	?	✓	?	?	?	?	?	High-performance consumer devices, wearables and enterprise	Done
SiFive's P870	Commercial	?	?	?	?	?	?	?	High performance consumer applications and datacenter	WiP
Ventana's Veyron V1	Commercial	?	?	?	?	?	?	?	Data center, edge, and general computing applications	Done
Frontgrade Gaisler's NOEL-V	Commercial	Sv32(x4) & Sv39(x4)	✓	Unified context cache + three separate TLBs: 1S/2S trans., guest trans. and MSI trans.	✗	WSI & MSI	✓	✓	Space (satellites, rovers, and other space-bound systems)	Done
CVA6 (Our Work)	SHL-2.1	Sv39(x4)	✓	IOTLB for 1S/2S trans., DDTC and PDTC	✗	WSI & MSI	✓	✓	General use and embedded platforms	Done

Table 2.5: Features and status of existing and ongoing RISC-V IOMMU implementations:

(✓) Supported; (✗) not supported; (?) no information available.

Companies deeply rooted in the RISC-V ecosystem, such as SiFive and Ventana, have integrated RISC-V IOMMU IPs into their latest processor series. These IOMMUs mainly aim at high-performance and general computing platforms, serving applications such as data centers, consumer devices, and edge computing. However, they are closed-sourced and exclusively accessible through commercial licenses. Consequently, almost no information about the architectural features of these IPs can be found publicly. Frontgrade Gaisler implemented a RISC-V IOMMU IP as part of their GRLIB IP library and integrated it into the NOEL-V processor, primarily aimed at space applications. This implementation supports the *Sv32/Sv32x4* and *Sv39/Sv39x4* translation schemes and includes four IOATCs, a hybrid interrupt generation mechanism, and an HPM. Nevertheless, as with previous implementations, NOEL-V is available under commercial licenses. On the open-source front, Rivos Inc. initially made available the RTL code of a RISC-V IOMMU during the development of the standard specification, intending to provide a fundamental reference for the RISC-V community. However, this endeavor did not progress further, and was last updated a year before the ratification of the RISC-V IOMMU specification, remaining far from materializing into a compliant implementation.

This work addresses the lack of a publicly available RISC-V IOMMU reference implementation by providing an open-source, functional, and parameterizable IOMMU IP compliant with the ratified RISC-V IOMMU specification v1.0. The main goal is to serve as a baseline reference for RISC-V practitioners, designers and academics, enabling a quick integration into general-purpose or embedded platforms. Moreover, it serves as a starting point for creating more advanced IPs tailored to high-performance applications. Furthermore, no existing commercial-based efforts have committed into a comprehensive design space exploration of the RISC-V IOMMU regarding hardware costs and translation latencies. We intend to leverage our implementation to conduct this study to propose alternative designs and optimizations to improve the IOMMU IP and the underlying RISC-V IOMMU standard specification.

Chapter 3

RISC-V IOMMU Specification

The RISC-V IOMMU standard specification, which began development in late 2021 with the formation of the RISC-V IOMMU Task Group, was officially ratified as v1.0 in June 2023. It outlines a set of mandatory and optional features, allowing implementations to adapt to specific application requirements while ensuring that all essential features defined by the standard are included. Additionally, the specification describes the in-memory data structures used to perform address translation, defines the software interface employed for programming and monitoring the IOMMU IP, and provides behavioral insights for address translation and other IOMMU-specific operations. This chapter summarizes the most relevant aspects of the RISC-V IOMMU specification, from now on referred to as "*the specification*." For more details about the specification, we refer the most interested readers to [11].

3.1 Address Translation

The specification introduces two memory-based data structures that store context information: The *Device Context* (DC) and the *Process Context* (PC). The DC associates a device with an address space and holds other per-device parameters the IOMMU uses to configure and perform address translation. It resides in main memory, in a one-, two-, or three-level radix-tree data structure named *Device Directory Table* (DDT), which is indexed using a unique device identifier: the *device_id*. Depending on whether the IOMMU supports MSI address translation, the DC may be 32-byte width (*base format*) or 64-byte width (*extended format*). As depicted in Figure 3.8, the extended format contains three additional MSI-related fields and one reserved doubleword.

Similarly, the PC associates a process with a virtual address space in devices that support multiple process contexts (e.g., PCIe devices). It holds controls for first-stage translation and is located in main memory, in a one-, two-, or three-level radix-tree structure named *Process Directory Table* (PDT), indexed

by a unique process identifier: the *process_id*. If a device does not support multiple process contexts, first-stage translation data is held in the DC. Otherwise, the DC retains the base address of the PDT.

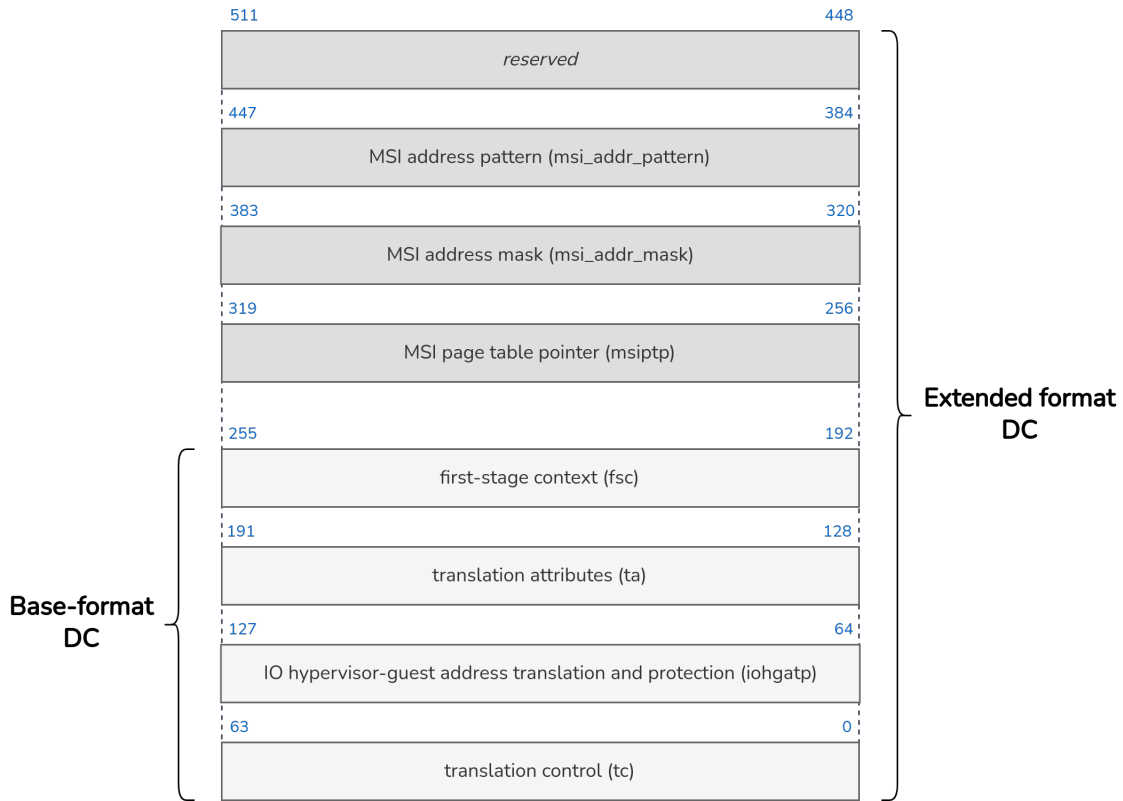


Figure 3.8: RISC-V IOMMU device context formats (adapted from [11]). The extended format includes three additional field the IOMMU uses to carry out MSI translation.

Context data is used to locate address mappings in memory, i.e., first and second-stage PTs. The IOMMU employs identical virtual memory schemes, PT formats, and two-stage address translation process as defined in the RISC-V privileged specification [17]. Hence, PTs used by the CPU's MMU can be simultaneously utilized by the IOMMU, reducing memory management complexity. The virtual memory scheme employed by the IOMMU may be configured individually per device, allowing the same IOMMU to perform two-stage, single-stage, or *bare* translations for different devices.

3.1.1 IO Address Translation Caches

Address translation within the IOMMU introduces performance implications for DMA transactions. The access time may be lengthened by the time required for walking memory to locate context and mapping data structures, and determine the final SPA. To reduce such added latency, the specification supports IOATCs to store translation-related data in hardware structures with faster access times. First and second-

stage PT data is stored in the IOTLB, while DCs and PCs are cached in the DDTC and PDTC, respectively. The IOMMU supports software commands to synchronize the IOATCs with their corresponding memory-resident data structures when modified (see subsection 3.3.2). Additionally, the specification defines two auxiliary tags to avoid aliasing conflicts when performing operations in the IOTLB: the *Guest Soft-Context Identifier* (GSCID) is configured by hypervisors to identify VMs and the *Process Soft-Context Identifier* (PSCID) is defined by guest or host OSes to identify process address spaces.

3.2 MSI Translation

When a DMA-capable device is configured directly by a guest OS, MSIs from this device are expected to be targeted to a virtual *Incoming Message-Signaled Interrupt Controller* (IMSIC) within the guest VM, using GPAs that are inappropriate and unsafe for the real machine. The RISC-V IOMMU specification includes optional support to redirect MSIs from guest-controlled devices to the corresponding destination. To support this capability, the architecture adds to the DC an MSI address mask and address pattern, used together to identify pages in the guest physical address space that are the destination of MSIs and the SPA of an MSI page table for translating MSIs from the device.

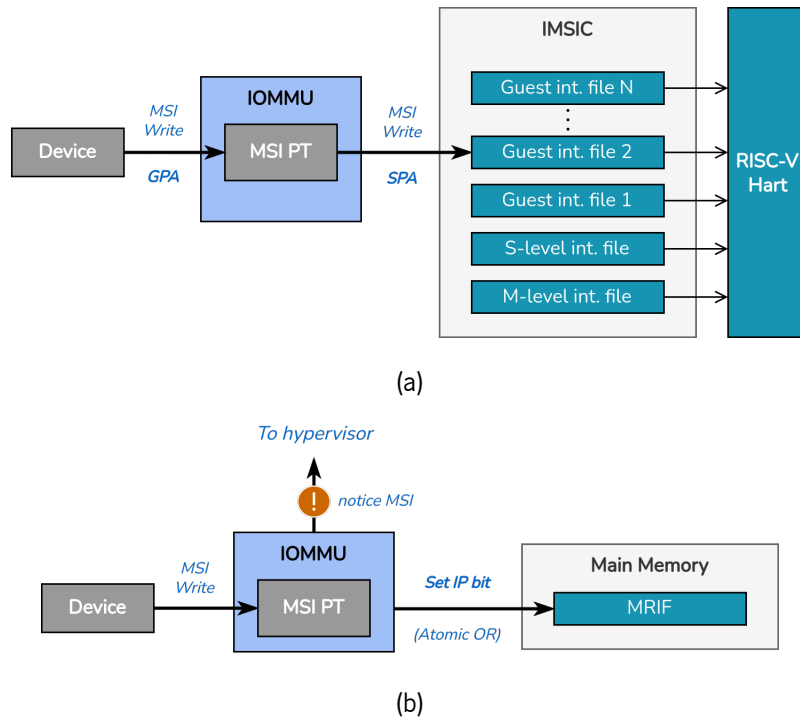


Figure 3.9: MSI translation modes supported by the RISC-V IOMMU:

(a) Pass-through mode; (b) *Memory-Resident Interrupt File* (MRIF) mode. (Adapted from [11]).

The *Advanced Interrupt Architecture* (AIA) specification [46] defines two modes for MSI translation, each associated with a different MSI PT format. In the first mode, known as *pass-through mode* (Figure 3.9a), the destination SPA represents the physical address of a guest interrupt file within an IMSIC. Alternatively, in the *MRIF mode* (Figure 3.9b), the MSI gets directed to an MRIF located in the main memory instead of within an IMSIC. In this later mode, the IOMMU records each MSI by setting an interrupt-pending bit stored within the MRIF. After each MSI is recorded, the IOMMU also sends a notice MSI to the hypervisor to inform it that the MRIF contents may have changed.

3.3 Software Interface

An IOMMU requires a means for software elements, such as the operating system or the hypervisor, to interact with the IOMMU hardware. This interface is crucial for configuring and managing address translation and access control for DMA-capable devices. Additionally, it enables software to track IOMMU activities and respond to events efficiently. The specification defines three interface subsystems for this purpose: (i) A **memory-mapped register interface** used by software to discover IOMMU capabilities, configure the IOMMU hardware, and monitor internal modules; (ii) three **memory-based circular buffers** (queues), each one with a *head* index controlled by the consumer and a *tail* index controlled by the producer; and (iii) an **interrupt generation subsystem** used by the IOMMU to request software service.

3.3.1 Memory-mapped Register Interface

The memory-mapped programming interface of the RISC-V IOMMU resides within a 4-kiB page of physical address space. As some IOMMU features are optional, implementing some registers related to these features is also optional. Table 3.6 summarizes the registers within the programming interface.

3.3.2 In-memory Queue Interface

The specification defines three memory-based queues to enhance hardware-software communication: (i) The CQ, (ii) the FQ, and (iii) the PQ. The size and location of each queue are established by software and conveyed to the IOMMU hardware through the base register of the queues, as illustrated in Figure 3.10. The CQ is used by software to issue commands to the IOMMU hardware. In the current version of the specification, four types of standard commands are defined (refer to Table 3.7): two for IOATC invalidation, one for software synchronization, and an optional type for PCIe features. Additionally, a reserved encoding

Register	Size (bits)	Description	Optional?
capabilities	64	IOMMU hardware capabilities	No
fctl	32	Features control	
ddtp	64	Device Directory Table Pointer and IOMMU/DDT mode	
cqb, cqh, cqt	64, 32, 32	Command Queue base and indexes	
fqh, fqh, fqt	64, 32, 32	Fault Queue base and indexes	
pqb, pqh, pqt	64, 32, 32	Page-request Queue base and indexes	If <i>capabilities.ATS</i> = 0
cqcsr	32	Command Queue control register	No
fqcsr	32	Fault Queue control register	
pqcsr	32	Page-request Queue control register	If <i>capabilities.ATS</i> = 0
ipsr	32	Interrupt pending status register	No
iocntovf	32	Hardware Performance Monitor counter overflow	If <i>capabilities.HPM</i> = 0
iocntinh	32	Hardware Performance Monitor counter inhibits	
iohpmcycles	64	Hardware Performance Monitor cycles counter	
iohpmctr1-31	64	Hardware Performance Monitor event counters	
iohpmevt1-31	64	Hardware Performance Monitor event selector	
tr_req_iova	64	Translation request IOVA	If <i>capabilities.DBG</i> = 0
tr_req_ctl	64	Translation request control	
tr_response	64	Translation request response	
icvec	64	Interrupt cause-to-vector register	No
msi_cfg_tbl	64, 32, 32	MSI configuration table (addr, data, ctl)	If <i>capabilities.IGS</i> = 1

Table 3.6: RISC-V IOMMU memory-mapped register interface. Some registers are optional depending on the inclusion of specific features, reported through the *capabilities* register.

space may accommodate custom commands supported by particular implementations.

Faults and events arising from the processing of transactions are communicated via the FQ. Each entry holds information related to the request (e.g., transaction type, IOVA and *device_id*) and a code indicating the underlying cause. Whenever a new record is written into the queue, the IOMMU generates an interrupt to alert software that a fault or event occurred. Subsequently, software "consumes" FQ data to find fault details. The PQ, meanwhile, acts as an interface to the IOMMU for reporting PCIe ATS *Page Request* and *Stop Marker* messages to software. Its implementation is optional if PCIe services are not implemented (i.e., *capabilities.ATS* = 0).

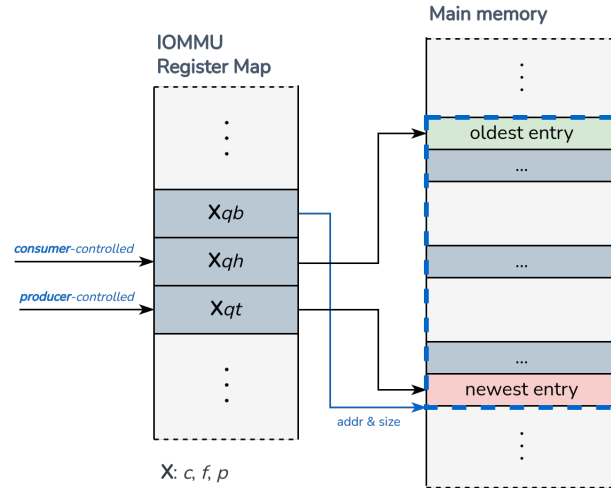


Figure 3.10: RISC-V IOMMU memory-based queue model (adapted from [11]). The head (X_{qh}) represents the next entry to be consumed. The tail (X_{qt}) points to the place where the next entry will be written, and the base (X_{qb}) holds the size and base physical address of the queue.

3.3.3 IOMMU Interrupt Generation

The specification outlines support for two interrupt generation mechanisms: WSI and MSI. At least one of these IGS mechanisms must be obligatorily implemented, while the inclusion of the other remains optional. The *icvec* register selects the interrupt vector for a specific interrupt source. It determines the interrupt wire that is set when using WSIs or the destination address and data of the MSI to be sent when using the MSI mechanism. Software can determine the source of an incoming interrupt reading the *ipsr* register.

The IOMMU may generate interrupts to request software service under various circumstances: (i) when any memory-based queue suffers a malfunction, (ii) upon the completion of an *IOFENCE.C* command, (iii) from the FQ or the PQ when the IOMMU hardware writes a new entry, and (iv) to signal a counter overflow in the HPM. It's possible to incorporate additional sources or causes of interrupts tailored to a specific implementation.

3.4 Miscellaneous Features

Depending on the intended application of the platform where integration is planned, the IOMMU may incorporate complementary optional features.

Command	Function
IOTINVAL.VMA	Invalidate first-stage page table data cached in the IOTLB
IOTINVAL.GVMA	Invalidate second-stage page table data cached in the IOTLB
IODIR.INVALID_DDT	Invalidate DDT entries cached in the DDTC and associated PDT entries cached in the PDTC
IODIR.INVALID_PDT	Invalidate PDT entries cached in the PDTC
IOFENCE.C	Signal completion of all previous fetched commands. Order memory accesses from DMA devices connected to the IOMMU
ATS.INVALID	Send an <i>Invalidation Request</i> message to a specific PCIe device function
ATS.PRGR	Send an <i>Page Request Group Response</i> message to a specific PCIe device function

Table 3.7: RISC-V IOMMU standard commands. Support for ATS commands is optional if *capabilities.ATS* = 0.

3.4.1 PCIe ATS and PRI

PCIe is an interface standard for connecting high-speed components, serving as the common motherboard interface in personal computers for graphics cards, sound cards, hard disk drives, host adapters, SSDs, Wi-Fi, and Ethernet hardware connections. ATS extends the PCIe protocol to support an address translation agent that translates DMA addresses to cached addresses in the device. The address translation cache in the device reduces the processing load on the translation agent, enhancing system performance. When an endpoint device determines that it requires access to a page for which the ATS translation is unavailable, it sends a *Page Request* message (through PRI services) to request the mapping of the page into system memory. The RISC-V IOMMU offers optional support for these two services, also enabling the integration of IOMMU hardware as a PCIe function within the PCIe bus. For a more comprehensive understanding of the PCIe protocol and its associated services, we recommend referring to the official PCIe specification document [47].

3.4.2 Hardware Performance Monitor

To gather performance-related data, the specification includes optional support for an HPM module. This subsystem includes (i) a 63-bit clock cycles counter (*iohpmcycles* register), (ii) up to 31 64-bit event

counters with event selection and ID-based filtering (*iohpmctr* and *iohpmevt* registers), (iii) counter overflow and inhibition control (*iocntovf* and *iocntinh* registers), and (iv) per-counter interrupt generation control upon overflow. The specification defines the standard events outlined in Table 3.8.

Event ID	Event Description
0	Do not count
1	Untranslated requests
2	Translated requests
3	ATS requests
4	IOTLB miss
5	Device directory walks (DDTC miss)
6	Process directory walks (PDTC miss)
7	First-stage page table walks
8	Second-stage page table walks
9 - 16383	Reserved for future standard use

Table 3.8: RISC-V IOMMU standard HPM events (adapted from [11]).

3.4.3 Register-based Debug Interface

To support software debugging, the specification provides an optional register interface that may be used by software to explicitly request the IOMMU to translate a specific IOVA. The interface consists of three registers (*tr_req_iova*, *tr_req_ctl* and *tr_response*) used by software to program an IOVA and other inputs needed by the IOMMU to perform address translation. If the process stops due to errors, the faults are reported normally in the FQ, and the *tr_response* register is updated with a failure indicator.

3.5 Use Cases for Virtualization

In virtualized systems, the RISC-V IOMMU enables three different approaches for managing DMA devices. The first method, *device pass-through*, permits direct control of a device by a guest OS with minimal hypervisor intervention. In this scenario, the device is programmed with GPAs by the guest OS, and the IOMMU executes second-stage translation. Alternatively, a hypervisor may provide a virtual IOMMU facility to the guest OS to benefit from intra-guest protection using a first-stage page table. In this case, the

IOMMU carries out two-stage address translation. Lastly, a host OS or hypervisor may choose to retain a device for its usage, relying solely on first-stage address translation. Figure 3.11 describes this concept.

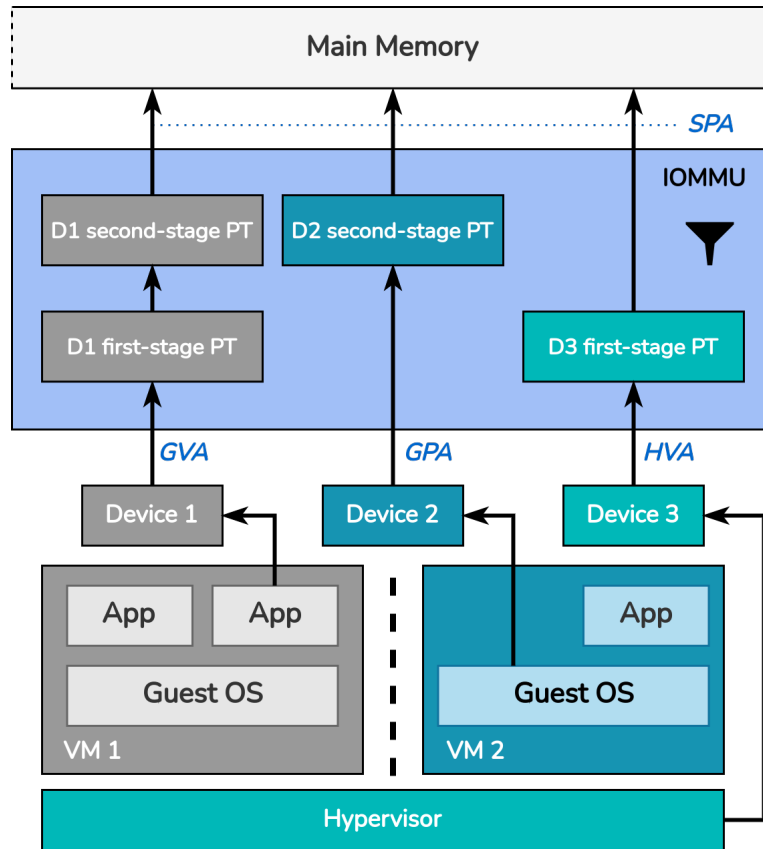


Figure 3.11: Usage models of the RISC-V IOMMU in virtualized systems (adapted from [6]). Device 1 is configured with GVAs as VM 1 benefits from intra- and inter-guest protection, and the IOMMU performs two-stage translation. Device 2 is *passed through* to the guest OS running on top of VM 2, so it utilizes GPAs. The hypervisor directly controls device 3. Thus, it is programmed with Host Virtual Addresses.

Chapter 4

RISC-V IOMMU Design

As mentioned in Chapter 3, the specification outlines a set of features, some being mandatory while others remain optional. The decision to include optional features depends on the specific use case and the properties of the target platform. In this sense, Table 4.9 provides an overview of the features incorporated into our design, distinguishing those considered mandatory by the specification. In the following section, we identify the architectural requirements that justify the inclusion of each optional feature.

RISC-V IOMMU Feature	Correlated Modules
Device Context and Process Context (★)	PTW, CDW, DDTC, PDTC
Two-stage address translation	PTW, IOTLB
Address translation caches	IOTLB, DDTC, PDTC
Memory-based queues (CQ and FQ) (★)	CQ Handler, FQ Handler, Regmap
MSI address translation and redirection	PTW, CDW, IOTLB, DDTC
Hardware performance monitor	HPM, Regmap
Interrupt generation support (★)	IGS, Regmap
Memory-mapped register interface (★)	Regmap

Table 4.9: Architectural IOMMU features considered for our use case, along with their correlated microarchitectural modules. Features marked with (★) are defined mandatory by the specification.

4.1 Architectural Requirements

The CVA6-based SoC designated for the integration of the IOMMU is endowed with hardware-assisted virtualization support. The purpose of integrating the IOMMU IP is to provide system-level memory protection. Consequently, the platform requires the inclusion of two-stage address translation support in

the IOMMU to provide spatial isolation: (i) between distinct VMs (*inter-guest* protection) via second-stage translation and (ii) between various applications running within the same guest OS (*intra-guest* protection) through first-stage translation. Additionally, to minimize translation latencies associated with memory walks and enhance overall system performance, we include the three IOATCs defined in the specification (i.e., IOTLB, DDTC, and PDTC).

To enable system-level interrupt virtualization support, the platform demands the integration of the RISC-V AIA (particularly the IMSIC module) and adding MSI translation support into the IOMMU. Since parallel efforts to incorporate the RISC-V AIA into the target SoC are in progress, MSI translation support has been included in the IOMMU. Additionally, we add the HPM to collect valuable performance-related data required to evaluate the IOMMU IP further.

In contrast, specific optional features are irrelevant in our particular scenario. For example, the target CVA6-based SoC intended for IOMMU integration lacks PCIe capabilities at present. Consequently, it is not practical to include these features in our design, as they would unnecessarily consume hardware resources. As a result, the PQ and its associated registers are omitted from the software interface of the IOMMU IP.

4.2 Design Parameters

In order to create a highly adaptable and customizable IOMMU IP, we define a set of design parameters, as detailed in Table 4.10. The purpose of these parameters is to control the incorporation of some optional features and configure microarchitectural properties of internal IOMMU structures. Consequently, this approach enables the development of multiple IOMMU versions, encompassing different combinations of architectural features (mandatory vs optional) and microarchitectural attributes.

In the subsequent sections of this chapter, we describe the influence of each parameter at the design level. It is worth noting that although process context support is categorized as a mandatory feature, we have designed it to be parameterizable. This design flexibility saves hardware resources in scenarios where the target platform lacks PC support. Non-Pcie SoCs, such as the one we target, commonly lack this feature.

Parameter	Function	Possible values
InclIPC	Include process context support	0, 1
IOTLB_ENTRIES, DDTC_ENTRIES, PDTC_ENTRIES	Define the number of entries for all IOATCs	[1 - N]
InclMSITrans	Include support for MSI translation in the address translation process	0, 1
IGS	Define supported mechanisms for IOMMU interrupt generation	WSI, MSI, WSI & MSI
N_INT_VEC	Define the number of interrupt vectors used by the IOMMU	[1 - 16]
N_IOHPMCTR	Define the number of event counters in the HPM	[0 - 31]
InclBC	Include 4-kB boundary check logic for AXI transactions (see Section 4.6.3)	0, 1

Table 4.10: IOMMU design parameters.

4.3 Base Microarchitecture

Figure 4.12 presents the diagram block of the IOMMU's microarchitecture, showing internal modules and interfaces. Each module is tied to one or more architectural features and design parameters, and their correlation is shown in Table 4.9. For example, the memory-mapped register interface is affected by the inclusion of the HPM, as it consequently incorporates counter registers into the design (see Table 3.6).

We classify the microarchitectural components of the IOMMU based on their functions into three primary groups: (i) **Translation modules**, which play a crucial role in the address translation process, enabling memory access to locate data structures, or caching context/translation data; (ii) **software interface modules**, responsible for creating communication channels between the IOMMU hardware and software for specific purposes; and (iii) the **external interfaces**, used by the IOMMU to interact with other hardware subsystems within the platform. In the remaining subsections of this chapter, we describe the design of these functional blocks and their corresponding modules.

4.4 Translation Logic

The translation subsystem of the IOMMU comprises the memory walker modules, namely the PTW and the *Context Directory Walker* (CDW), along with the IOATCs. This subsystem is responsible for receiving DMA requests, locating the pertinent context data, conducting address translation (if needed), and re-

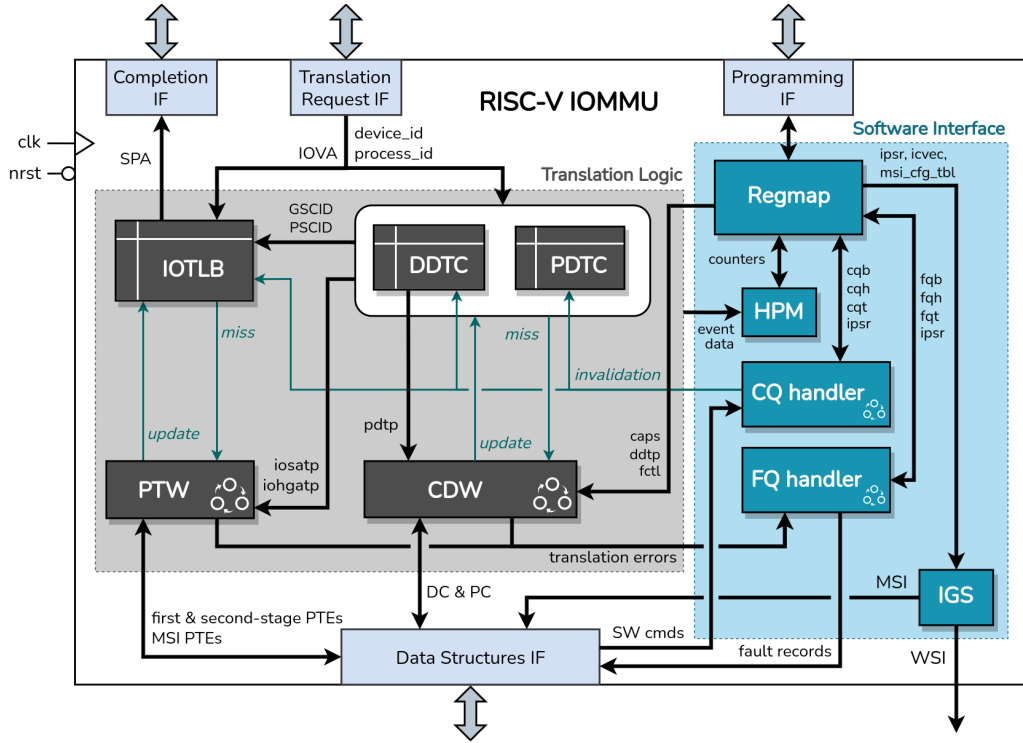


Figure 4.12: Base IOMMU microarchitecture:

(●) translation modules; (●) software interface modules; (●) external interfaces.

sponding the requested transaction. Furthermore, when MSI translation support is enabled (*InclMSITrans* = 1), this subsystem also handles the translation of MSI addresses. The IOVA translation process, as described in the specification, is synthesized in the flowchart of Figure 4.13. More complex subprocesses, such as memory walks, are detailed for each module in the following subsections. The translation process ends with the final SPA upon successful completion. In contrast, if a fault occurs in any subprocess, an error completion is generated.

4.4.1 Context Directory Walker

The CDW module is used to locate both the Device Context and, if available, the Process Context. When the IOMMU is enabled, the CDW is triggered upon a DDTC/PDTC miss to walk main memory, find the corresponding context data structure, and subsequently update the pertinent cache. To interact with memory, the CDW module utilizes the external memory interface of the IOMMU, explained in detail in section 4.6.

Before delving into the design specifics of the CDW module, it is convenient to explain how context directory walks are conducted in main memory. Figure 4.14 illustrates this procedure and the different

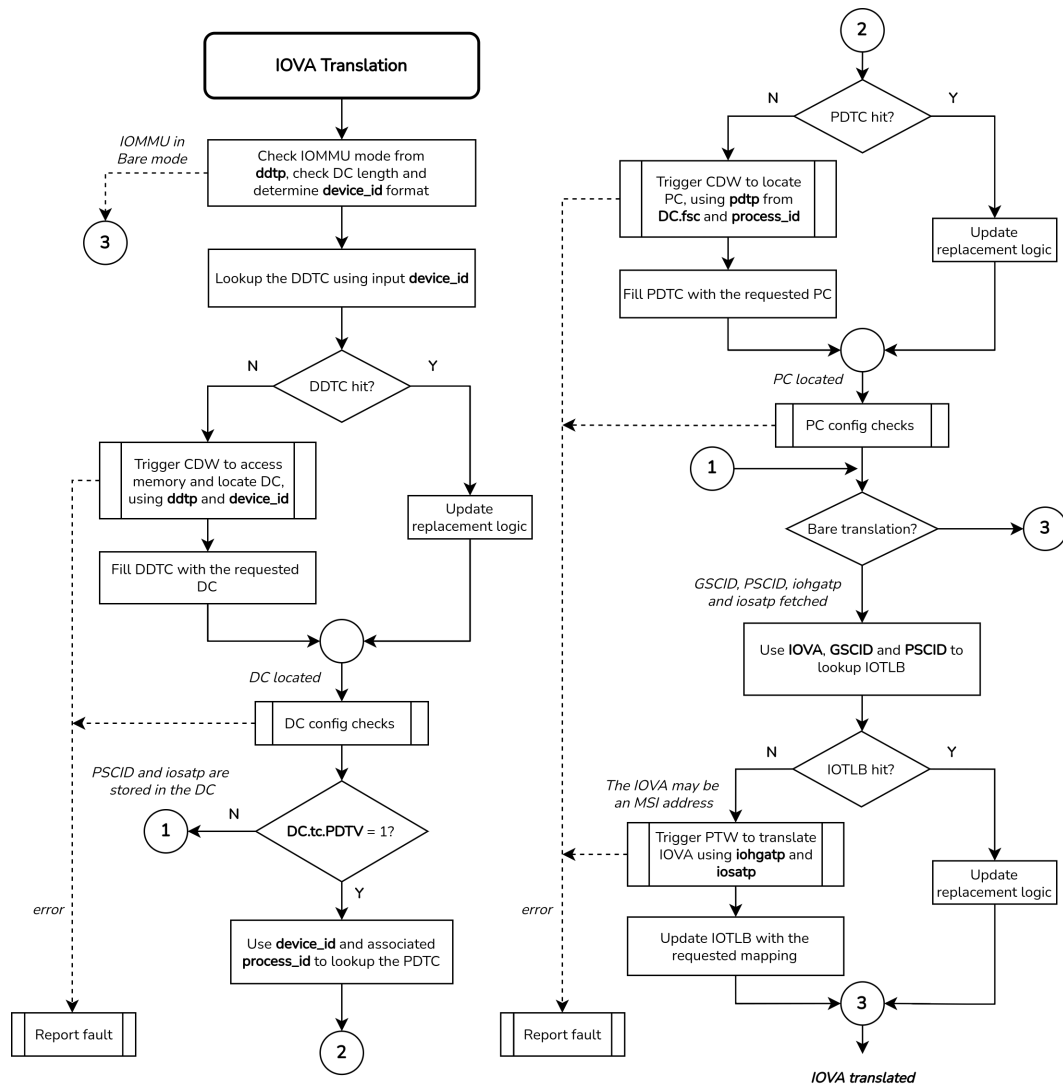


Figure 4.13: IO virtual address translation process.

paths it may take. Upon a DDTC miss, the CDW must walk memory using the provided *device_id* to locate the required DC. Depending on whether MSI translation is enabled (as defined by the *IncMSITrans* design parameter), the *device_id* is segmented differently. The base physical address of the DDT is stored in the *ddtp* register within the IOMMU. This register also determines the IOMMU mode (e.g., Off, Bare, On) and the number of levels of the DDT. According to the number of levels specified in the *ddtp* register, one, two, or three memory accesses are needed to locate each DC. Additionally, the number of accesses required to fetch the entire DC depends on whether MSI translation is enabled, as the DC width increases when this feature is active.

Once the DC is fully loaded, it becomes possible to check if any PC is associated with the transaction. As depicted in the flowchart in Figure 4.13, if *DC.tc.PDTV* = 0, no PC is associated. Consequently, the memory walk concludes, and the DC contains all the information required for first-stage, second-stage, and

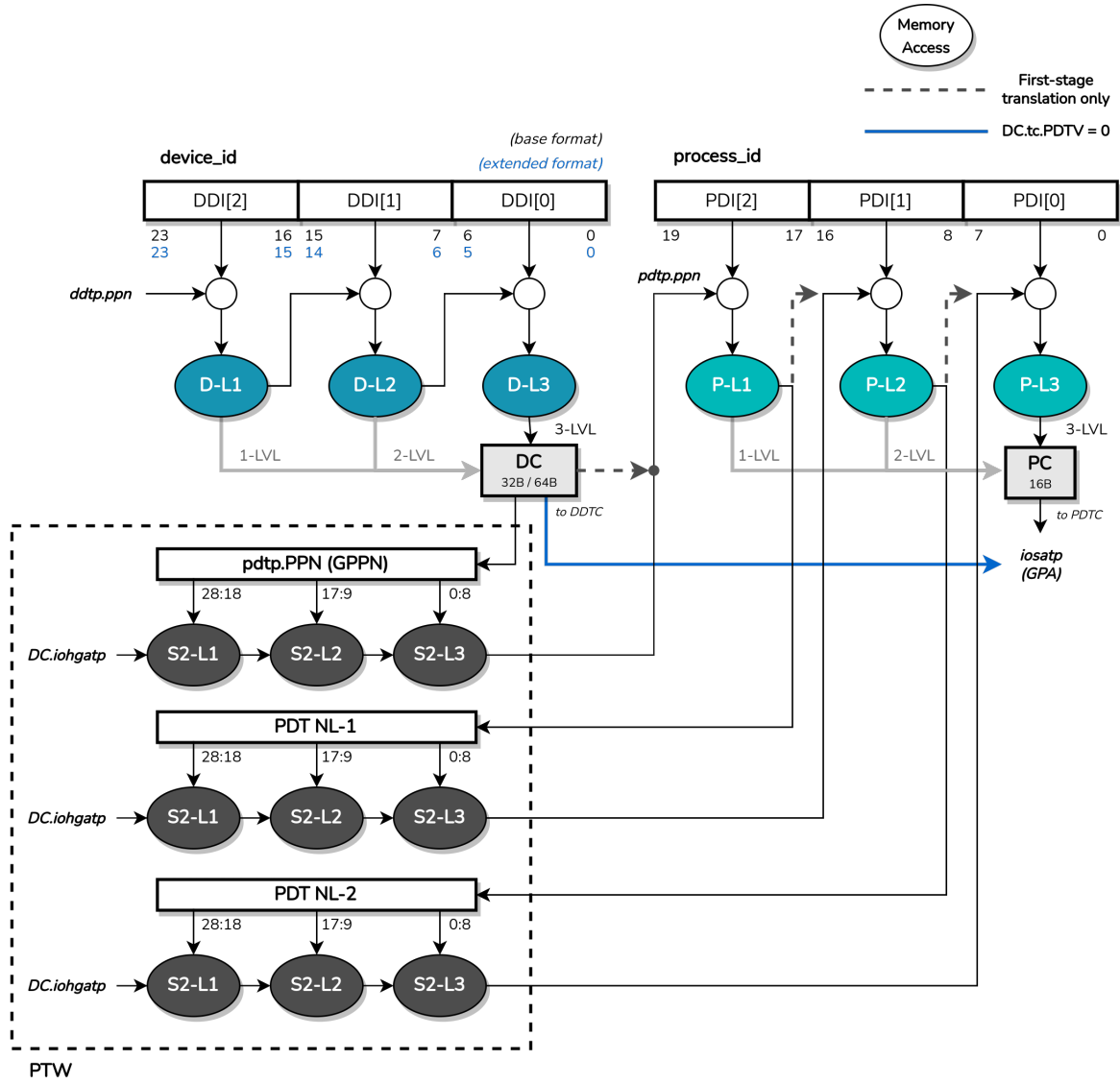


Figure 4.14: Context directory walk procedure. Three factors determine the flow of the walk: (i) The presence of a PC, (ii) second-stage translation, and (iii) the number of levels of each table.

MSI translation. Conversely, if $DC.tc.PDTV = 1$, and the required PC is not present in the PDTC; the CDW must walk memory using the input *process_id*. In this scenario, the DC holds the base address of the PDT and the number of PDT levels. If second-stage translation is enabled, the PTW module must previously translate the base address, and the number of PDT levels determines how many memory accesses are required to locate the PC. Unlike the DC, the size of the PC remains fixed at 16 bytes.

Guided by the context directory walk procedure, we devised a *Finite State Machine* (FSM) to drive the operation of the CDW module. The state transition diagram of this FSM is depicted in Figure 4.15. Among these states, *LEAF* and *GUEST_TR* are notably influenced by design parameters, particularly by MSI translation support (*InclMSITrans*) and process context support (*InclIPC*). Below, we explain the function

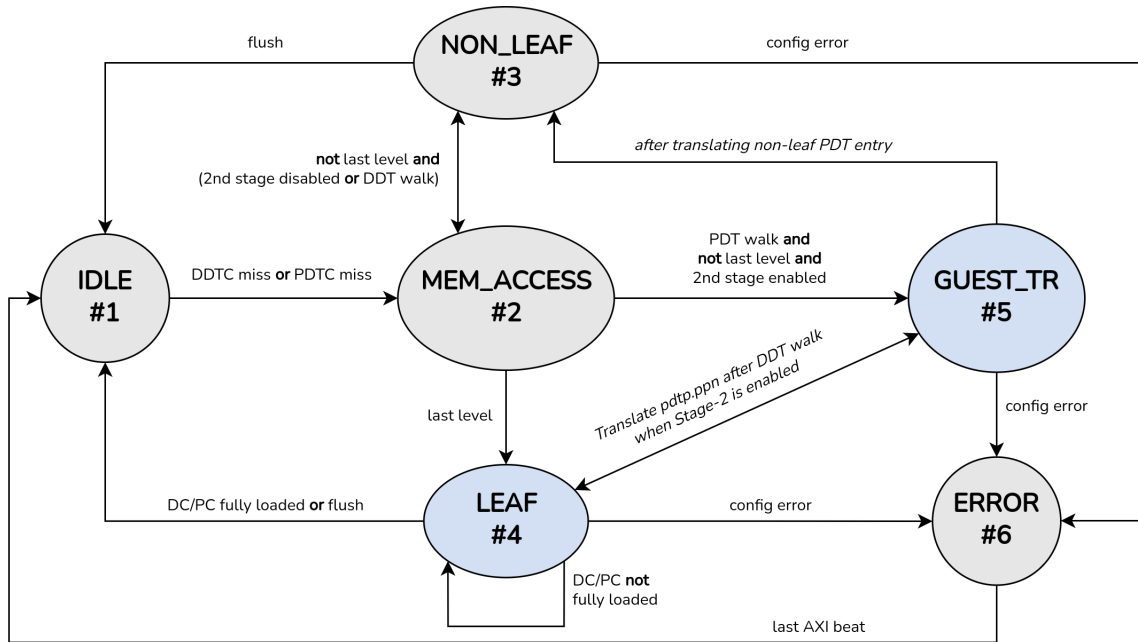


Figure 4.15: Finite state machine of the context directory walker. The states filled with (●) are the most affected by design parameters.

of each state.

1. **IDLE:** This state monitors potential DDTC/PDTC misses and records the number of table levels required to walk in memory. It determines the address of the first memory request;
2. **MEM_ACCESS:** Requests access to main memory and awaits for a response. The next state is decoded based on the walk type, current walk level, and the situation regarding second-stage translation;
3. **NON_LEAF:** When the walk level is not the last, this state processes non-leaf entries, performs configuration checks, and determines the address of the memory request for the next walk level;
4. **LEAF:** This state handles the loading and validation of leaf entries (DC/PC). Given the maximum 64-bit width of the target platform's memory bus, data is loaded iteratively. In the last iteration, the DDTC and PDTC caches are updated correspondingly;
5. **GUEST_TR:** Triggers the PTW module to perform implicit second-stage address translations. This is required when second-stage translation is enabled for the address found in non-leaf PDT entries and for the *DC.fsc.pdtp* field;
6. **ERROR:** This state signals the occurrence of an error and propagates the cause code.

The *LEAF* state is predominantly influenced by MSI translation support, as this feature increases the number of iterations needed to load the complete DC into the CDW module. Furthermore, when second-stage translation and PC support are both enabled, two additional steps come into play: (i) the base address of the PDT is translated prior to updating the DDTC, and (ii) as addresses configured in non-leaf PDT entries hold GPAs rather than physical addresses, they must be translated when walking the PDT. These implicit translations are triggered by the *GUEST_TR* state in the PTW module. When *InclPC* = 0, no implicit second-stage translations are needed, leading to the exclusion of this state from the design.

4.4.2 Page Table Walker

The PTW module is responsible for translating IOVAs to SPAs using the page tables located in main memory. In a virtualized environment, first-stage PTs are set up by a guest OS or the host OS/hypervisor when it possesses exclusive device control. As demonstrated in Figure 4.14, the base address of the PT is stored in the *iosatp* field of the DC when *InclPC* = 0. Alternatively, it is stored in the PC when *InclPC* = 1. The hypervisor configures second-stage PTs, and the base physical address of each PT is stored in the *iohgap* field of the DC. Additionally, the PTW module is also responsible for MSI translation when the input IOVA is identified as the address of a virtual interrupt file. For each device, the hypervisor configures an MSI page table in main memory and stores the base physical address in the DC.

Similar to the CDW, before delving into the PTW module's design, we will explain the process of executing nested memory walks in main memory. This process is subsequently translated into an FSM that governs the operation of the PTW module.

The CVA6 core implements the *Sv39x4* virtual memory scheme, as defined by the RISC-V privileged specification [17]. Consequently, our PTW is designed to conduct address translations following with this scheme. *Sv39x4* defines the format of PTEs and the two-stage address translation process, detailed in Figure 4.16. Depending on the configuration of first-stage and second-stage translation and the utilization of superpages, the translation process may follow different paths. If both stages are enabled, the input address is a GVA, and first-stage PTs are configured with GPAs instead of real physical addresses. The process is called "*nested*" because addresses derived from *iosatp* (GPA_0 in Figure 4.16) or any first-stage PTE (GPA_1,2,3) must undergo second-stage translation before accessing physical memory. The translation process concludes upon locating the first-stage leaf PTE and performing the final GPA translation through second-stage PTs. Alternatively, when only one translation stage is active, memory accesses occur directly using the address stored in *iosatp* for first-stage translation or *iohgap* for second-stage translation. The RISC-V privileged specification defines three levels for PT walks, utilizing 4-kiB pages. However,

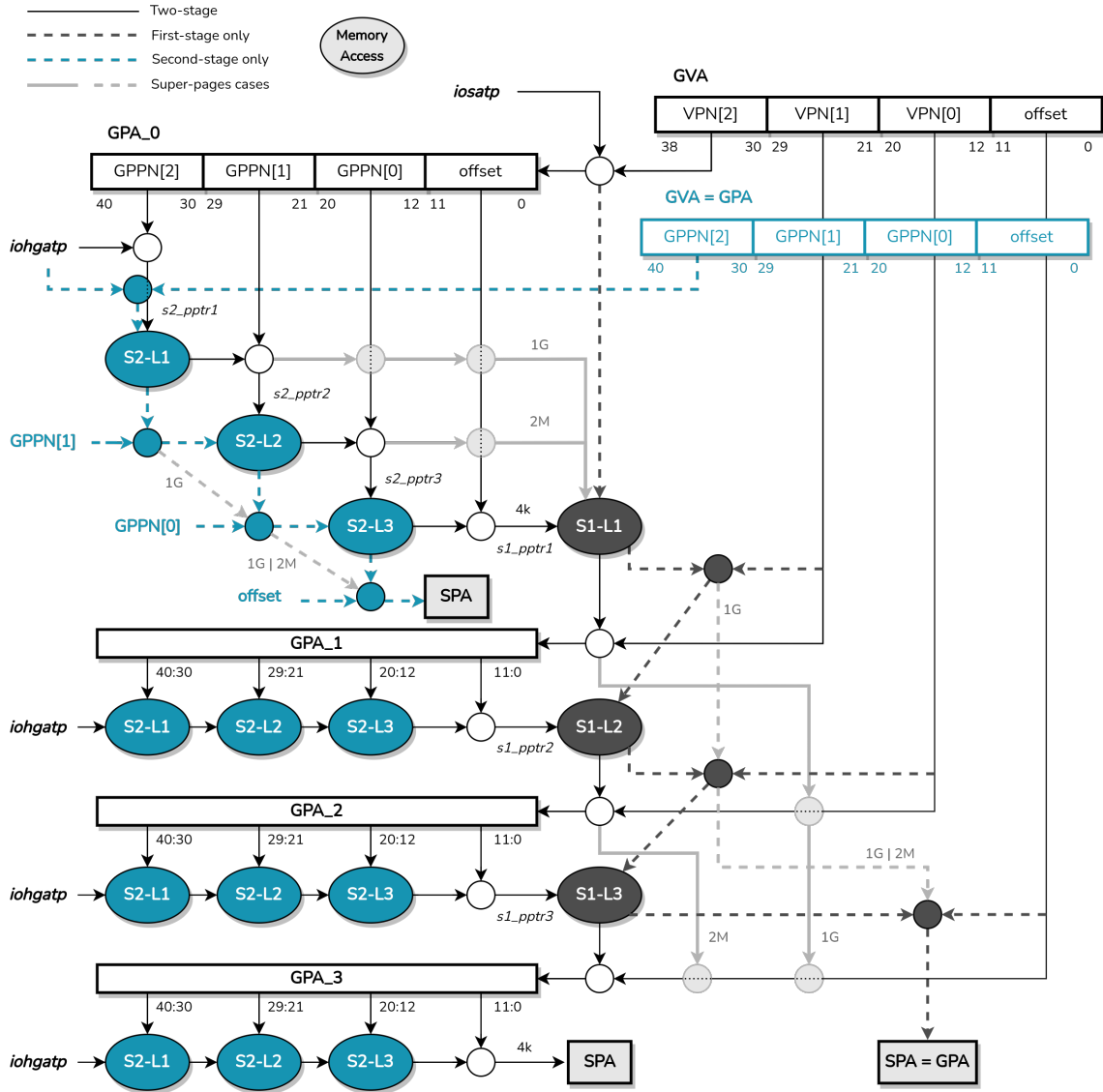


Figure 4.16: Sv39x4 nested memory walk. The walk flow depends on whether first and second-stage translations are enabled, and superpages are used.

it also allows for the configuration of 2-MiB and 1-GiB superpages. This configuration reduces the number of levels the PTW must walk to locate a leaf PTE, effectively expanding the range of addresses covered by this PTE.

We designed the FSM for the PTW module based on the nested translation process. The flow described in Figure 4.16 is translated into the state transition diagram of Figure 4.17. In addition to the five states of the diagram, we defined three internal stages to control the walk flow depending on which translation stages are enabled. **STAGE_2_INT** represents in-between second-stage translation of addresses derived from non-leaf first-stage PTEs. In a scenario without superpages, these addresses are GPA_0,1,2 in Figure 4.16. **STAGE_1** encompasses memory accesses to first-stage PTEs. When only first-stage translation

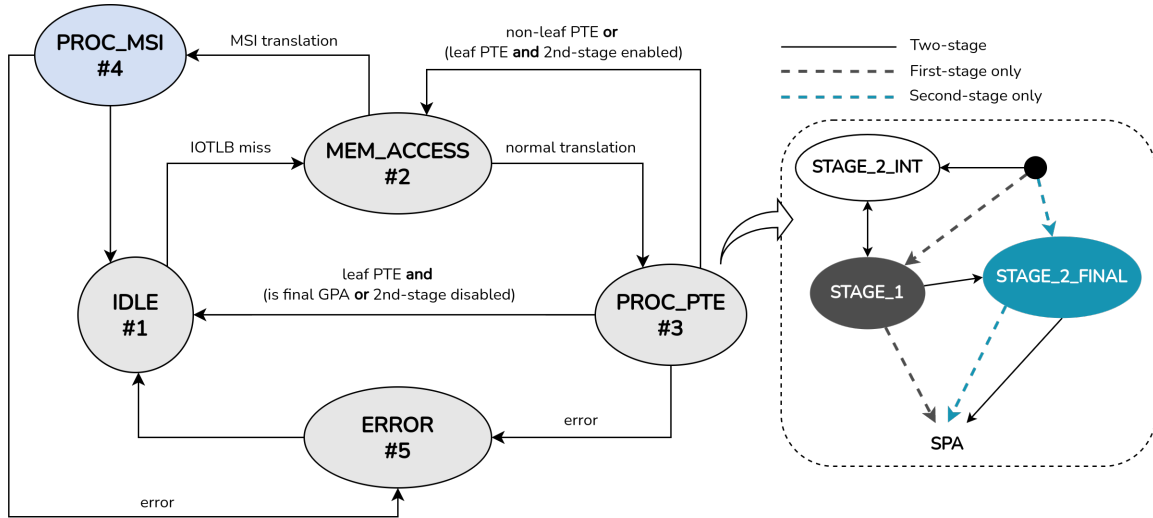


Figure 4.17: Finite state machine of the page table walker. Three internal stages are defined to control the walk flow in normal translations. State *PROC_MSI* is omitted if MSI translation is not supported.

is enabled, addresses derived from these PTEs are used directly to access physical memory, so this is the only stage employed. **STAGE_2_FINAL** corresponds to the second-stage translation of the last GPA (GPA_3 in Figure 4.16 if no superpages are found). The result is always the final SPA, and, if only second-stage translation is enabled, no other stage is employed. The operation of the main FSM is described next.

1. **IDLE**: This state checks for eventual IOTLB misses and defines the first walk stage. It also determines whether the input IOVA corresponds to an MSI address;
2. **MEM_ACCESS**: Requests access to main memory and awaits for a response;
3. **PROC_PTE**: PTEs associated with normal address translation are always processed in this state, according to the procedure described in Figure 4.16. At the end, the IOTLB is updated;
4. **PROC_MSI**: Processes MSI PTEs and updates the IOTLB with the translated address. MSI PTs are always flat, i.e., single-level. Thus, only one memory access is needed to translate the input address. This state is omitted in the design if MSI translation is not supported;
5. **ERROR**: Signals the occurrence of an error and propagates the cause code.

4.4.3 IO Address Translation Caches

Figure 4.18 provides a visual representation of the internal structure of an entry within each IOATC. In our approach, we store data from both the first and second-stage translations within the same IOTLB

entry, effectively combining information from both stages. This choice offers the advantage of reducing hardware costs by minimizing the need for tag comparisons during lookups. However, it requires merging the translation size of both stages into the smaller of the two to prevent potential aliasing issues in IOTLB lookups. Thus, for hypervisors to utilize superpages, guest OSes running on top of them must also support superpages. Additionally, MSI translations in the IOTLB are managed using a single identification bit (*is_msi*).

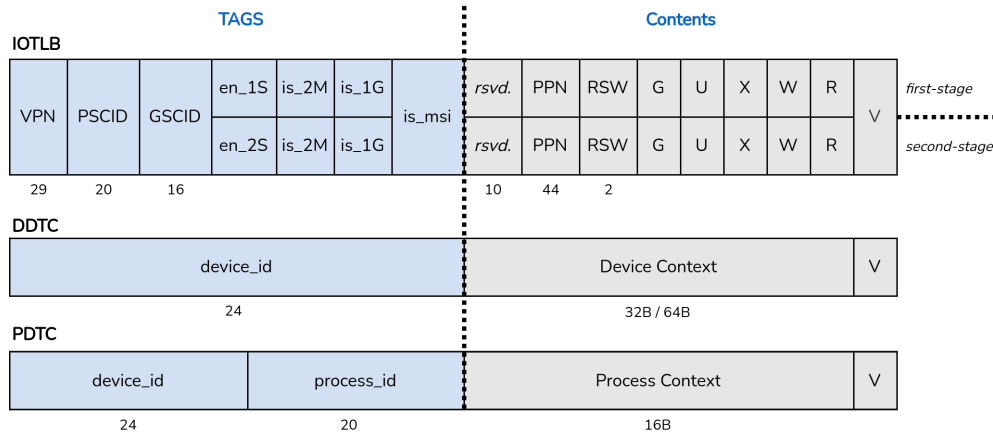


Figure 4.18: Structure of an entry in each IOATC.

To achieve maximum efficiency, the lookup logic in all IOATCs is designed to be fully combinational. Hence, data is instantly available to the external translation logic when cache hits occur. Updates are executed with one cycle of latency, following a *pseudo-least recently used* replacement policy to select the victim entry. Additionally, all IOATCs can be selectively flushed via software commands issued through the CQ. Each flush operation consumes a single clock cycle from the moment the corresponding signal is received from the CQ handler. Furthermore, the number of entries in each IOATC is determined by three design parameters (refer to Table 4.10). It is worth noting that the inclusion of MSI translation support significantly increases the size of each DDTC entry.

4.5 Software Interface

Within our IOMMU, the software interface subsystem encompasses the memory-mapped register interface, a handler module for each memory-based queue (CQ and FQ), the IGS module, and the HPM. The purpose of this subsystem is to provide communication channels between software executing on the processor and the IOMMU hardware. It is important to remark that PCIe-related features are intentionally excluded from the IOMMU design. As a result, the PQ is omitted, along with its associated registers. The

HPM is considered part of this subsystem, as it provides software a means to gather valuable performance-related data through the memory-mapped register interface.

4.5.1 Memory-mapped Register Interface

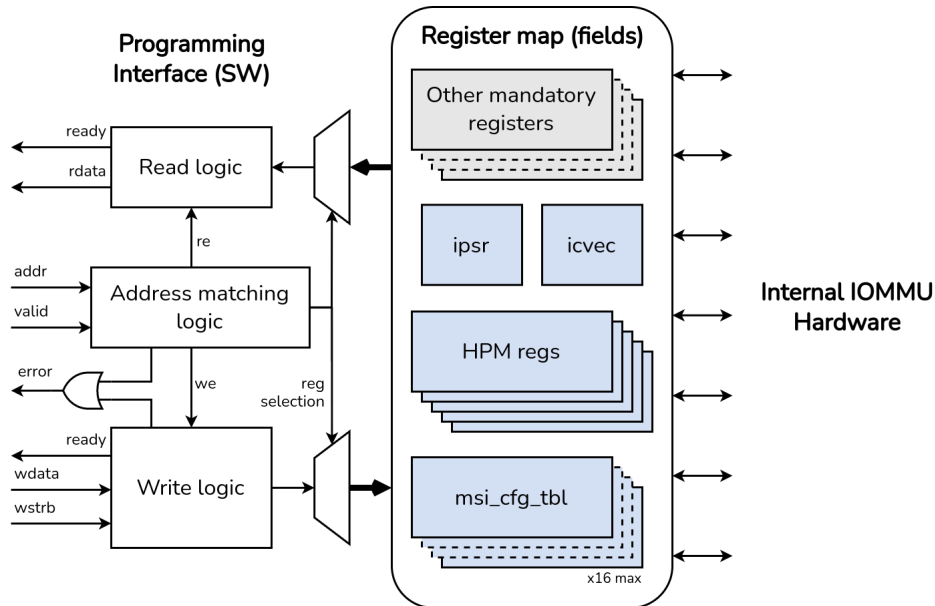


Figure 4.19: Core design of the IOMMU register map. Consisting of an array of register fields, registers are categorized based on the impact of design parameters as either () unaffected or () affected.

As previously mentioned, our IOMMU lacks support for PCIe capabilities, i.e., *capabilities.ATS* is always zero. Consequently, the PQ-related registers are absent from the memory-mapped interface (see Table 3.6). Similarly, our IOMMU does not support the debug register interface outlined in the specification. Therefore, the *tr_req_iova*, *tr_req_ctl*, and *tr_response* registers are omitted from the design. Furthermore, some registers are tied to IOMMU features influenced by design parameters. As a result, the inclusion, size, or format of these registers is conditioned.

Figure 4.19 illustrates the core design of the IOMMU register map, highlighting (in blue) the registers that are affected by design parameters. For instance, the *N_IOHPMCTR* parameter determines the number of HPM event counters. Thereby, it defines the number of *iohpmctr* and *iohpmevt* registers included in the design, as well as the number of valid bits in the *iocntovf* and *iocntinh* registers. When *N_IOHPMCTR* = 0, the entire HPM is omitted from the design (i.e., *capabilities.HPM* = 0), and the corresponding interrupt-related fields are excluded from the *ipsr* and *icvec* registers. Likewise, the *IGS* design parameter determines whether the *msi_cfg_tbl* registers are included in the design (only if *IGS* = MSI or *IGS* = BOTH). Lastly, the *N_INT_VEC* parameter defines the number of *msi_cfg_tbl* registers (up to 16) when

MSI generation is supported and the format of the *icvec* register.

The register map is accessible from two different sides: (i) via a simple register interface port (left), which the programming interface utilizes to enable software configuration of the IOMMU, and (ii) through direct bidirectional connections linking all registers to the internal hardware logic of the IOMMU (right). In the former interface, a single register is indexed using address-matching logic for reading and writing operations. To ensure harmonization in the representation of IOMMU registers and to provide finer granularity when transmitting register data to the internal logic of the IOMMU, we opted for a field-oriented register map design, i.e., a unified standalone storage module is employed to represent each register field, and the register map consists of an array of these fields. Nevertheless, it's important to note that software transactions always target entire registers. Consequently, a single transaction may span one or more fields, depending on the structure of the register.

Given that the smallest IOMMU register size is 32 bits and to fit into RV32 platforms, we designed the register map to support only 32-bit accesses. Thereby, register fields spanning across the 32-bit boundary are split into two separate instances. Figure 4.20 provides an internal view of each register field, with software-controlled signals on the left side and hardware-controlled signals on the right side. In order to prevent write conflicts between both interfaces, we prioritize software writes over internal hardware writes.

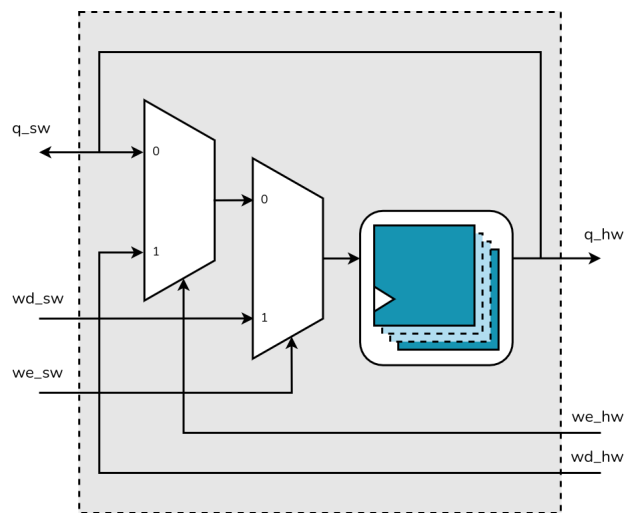


Figure 4.20: IOMMU register field design. Signals on the left side are employed by software via the programming interface, while the internal IOMMU hardware logic utilizes signals on the right side.

4.5.2 Command Queue Handler

The CQ handler module continuously monitors the index registers of the CQ. We use comparison logic to determine whether the index in the *cqt* register differs from that in the *cqh* register. In such a case, an FSM is activated to fetch and decode all pending commands until both indexes become equal (considering the CQ's size). It is worth noting that the CQ handler remains unaffected by any design parameter, resulting in identical behavior for all IOMMU configurations. Furthermore, since our IOMMU does not support PCIe features, the *ATS* commands outlined in the specification (refer to Table 3.7) are unsupported and considered illegal. Figure 4.21 illustrates the state diagram of the CQ handler. Following, we explain the flow of each state within the FSM.

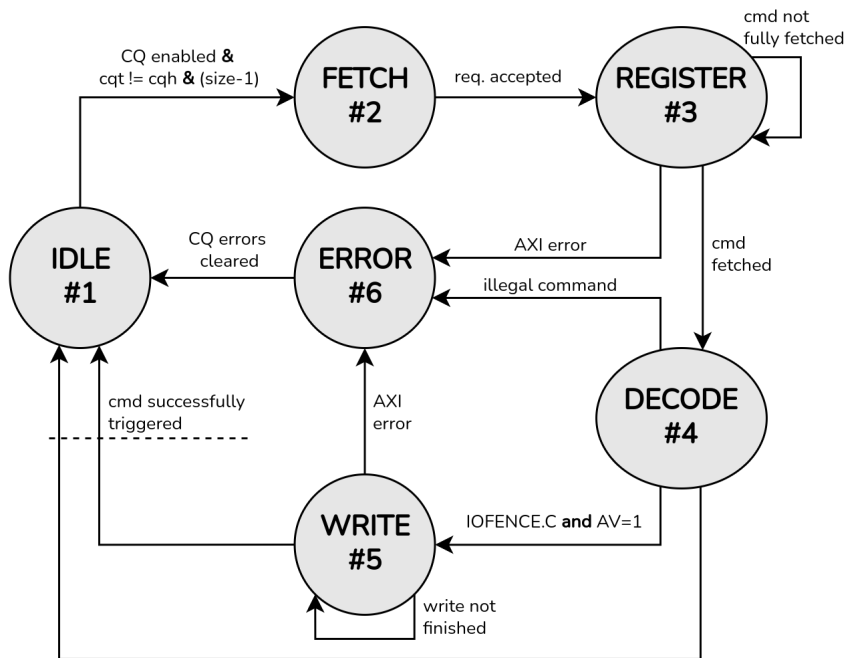


Figure 4.21: States diagram of the command queue handler. The *WRITE* state triggers a second-level FSM to perform AXI writes.

1. **IDLE:** This state compares both *cqt* and *cqh* registers. If the CQ is enabled and the value of both indexes is different after applying the corresponding CQ size mask, the FSM is triggered to fetch the next pending command in memory;
2. **FETCH:** Sends the first request to access to main memory and awaits for a response;
3. **REQUEST:** During normal operation, this state is executed twice to store the entire CQ entry, which spans 16 bytes. After fetching the whole command, the head index is incremented;

4. **DECODE:** In this state, the integrity of the fetched command is verified. Subsequently, it configures the command parameters and transmits the execution signals to the corresponding entity;
5. **WRITE:** This state is only reached when executing an *IOFENCE.C* command with the *AV* parameter set to 1. It triggers a second-level FSM that writes a specified 32-bit word to a 4-byte aligned address upon command completion. This second-level FSM is similar to that used in the write state of the FQ handler. For a more detailed explanation of this write mechanism, please refer to Section 4.5.3;
6. **ERROR:** This state signals an internal CQ error by setting the corresponding error bits in the *cqcsr* and *ipsr* registers and generating an IOMMU interrupt. When an error occurs, the FSM remains locked in this state until all CQ errors have been cleared.

4.5.3 Fault Queue Handler

Much like the CQ handler, we designed the FQ handler as an FSM that remains unaffected by any design parameter. However, there is a key distinction in this case: the IOMMU acts as the producer of the queue rather than the consumer. The first-level FSM is activated when the external translation logic signals a fault and if the FQ is enabled. Additionally, to prevent the loss of records when faults arrive at a rate faster than the IOMMU can write them to the FQ, we integrated a *fall-through* FIFO⁷. The state diagram of the FQ handler is depicted in Figure 4.22, and the operation of each state is explained ahead.

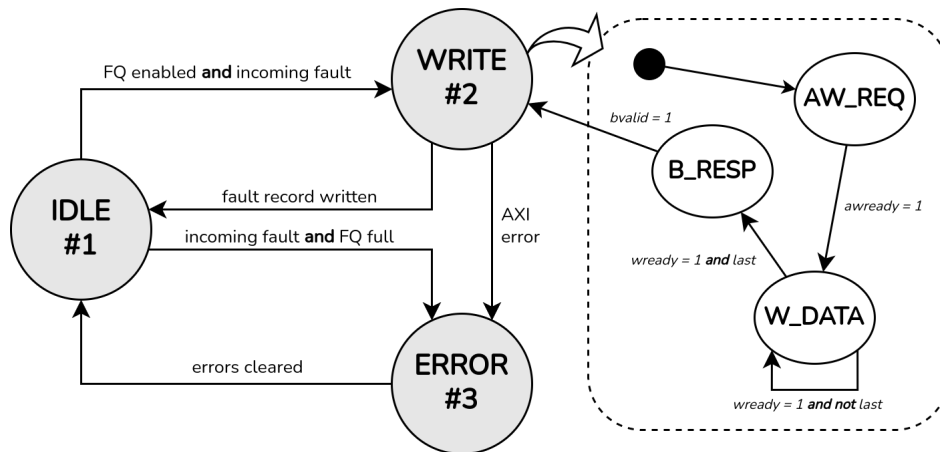


Figure 4.22: Fault queue handler's states diagram. The first-level FSM controls the operation of the FQ handler, whereas the second-level FSM is responsible for writing fault records into the FQ.

⁷When empty, a fall-through FIFO places input data instantly in the output port. If push and pop signals are set at the same time, data is not stored in the FIFO.

1. **IDLE:** This state monitors possible faults signaled by the external translation logic. When an incoming fault occurs with the FQ enabled, a record is generated containing event data registered based on the cause of the fault and the procedure outlined in the specification;
2. **WRITE:** This state activates a second-level FSM responsible for writing the 32-byte fault record into the FQ. This FSM executes the complete AXI4 write procedure, iterating four times in the W_DATA state to transmit all 32 bytes, using 8-byte beats. The CQ handler and the MSI generation module employ a similar mechanism.
3. **ERROR:** This state signals the occurrence of an internal FQ error by setting the corresponding error bits in the *fqcsr* and *ipsr* registers and generating an IOMMU interrupt. When an error occurs, the FSM remains locked in this state until all FQ errors have been cleared. Consequently, the report of translation errors is suspended.

4.5.4 Interrupt Generation Support

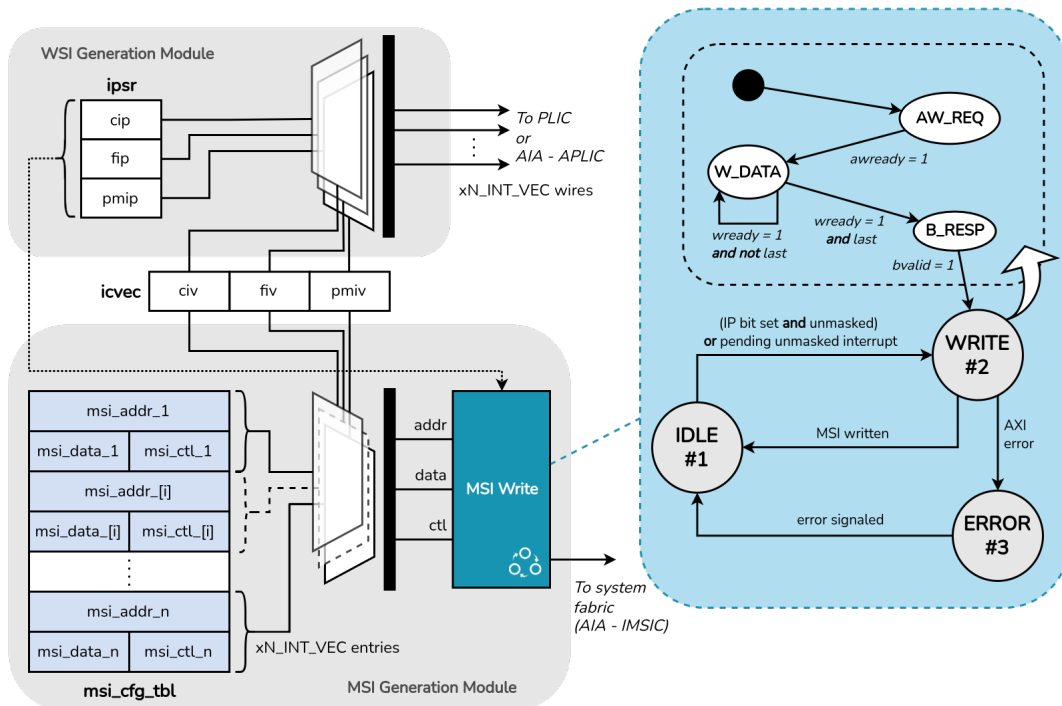
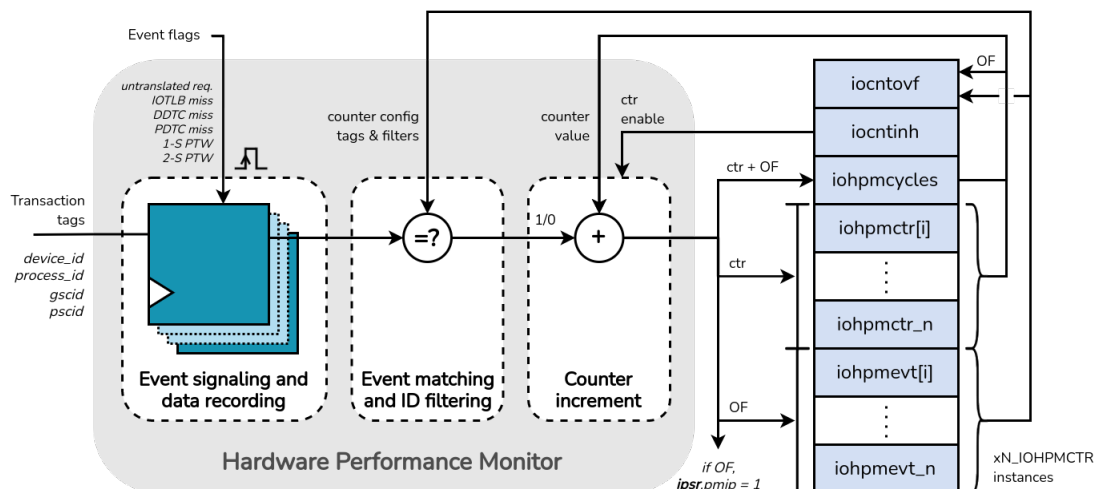


Figure 4.23: Interrupt generation subsystem design. Both mechanisms were designed modularly to ensure their independence from one another.

As mentioned in section 3.3.3, the specification defines two mechanisms by which the IOMMU can generate interrupts: WSI and MSI. The *IGS* parameter offers the capability to define at design time which

of these mechanisms are to be incorporated into the implementation. Figure 4.23 illustrates the overall design of the IOMMU interrupt generation subsystem. The WSI generation module routes the interrupt pending bits from the *ipsr* register to an external interrupt wire, indexed by the corresponding vector in the *icvec* register. These interrupt wires are directly connected to either the *Platform-Level Interrupt Controller* (PLIC) or the *Advanced Platform-Level Interrupt Controller* (APLIC)⁸ module of the platform. In contrast, the MSI generation module uses the *icvec* register as an index to a set of registers in the MSI configuration table (*msi_cfg_tbl*), selecting the corresponding destination address, data, and interrupt mask to write an MSI into the appropriate interrupt file within the platform's IMSIC⁸. The FSM executes the write operation depicted in Figure 4.23. It triggers when an interrupt pending bit is set in the *ipsr* register (provided the corresponding vector is unmasked) or when a masked vector with a pending interrupt is unmasked. The number of interrupt vectors (i.e., the number of interrupt wires and *msi_cfg_tbl* entries) is defined at design time by the *N_INT_VEC* parameter.



The HPM is composed of three main functional blocks, as illustrated in Figure 4.24: (i) event signaling and data recording; (ii) event matching and ID filtering; and (iii) counter increment. The first block receives event notifications through external signals (flags). Our design supports six types of events, outlined in Figure 4.24, and each event type has its dedicated flag. When an event occurs, indicated by the rising edge

of the corresponding event flag, the HPM registers all IOMMU transaction tags and records the event's ID. Then, in the second block, we compare the registered event data with the counters configuration (i.e., *iohpmevt* registers) to determine whether any counter is programmed for the occurred event. This comparison considers additional filters (i.e., ID filters), which are configured within the *iohpmevt* registers. In the final block, the HPM simultaneously increments all matching event counters (*iohpmctr* registers) that are not inhibited according to the *iocntinh* register. Additionally, the *iohpmcycles* counter is incremented each clock cycle unless inhibited. Upon an overflow in any event counter or in the *iohpmcycles* counter, the HPM sets the overflow flag (OF) for the corresponding counters and generates an interrupt by setting the *pmip* bit in the *ipsr* register. All bits in the *iocntovf* register are directly connected to the OF bits within the register map. Furthermore, the number of event counters (i.e., *iohpmctr* and *iohpmevt* registers) is defined at design time through the *N_IOHPMCTR* parameter. The entire HPM can be excluded from the design setting *N_IOHPMCTR* to zero.

4.6 External Interfaces

As shown in Figure 4.12, our IOMMU incorporates four external interfaces for system-level communication: (i) the **programming interface**, (ii) the **data-structures interface**, (iii) the **translation request interface**, and (iv) the **completion interface**. We selected the AXI protocol to develop these interfaces, given its flexibility and widespread adoption as a standard. The AXI standard defines a set of signals organized into channels (AR, R, AW, W, and B). For each interface, we bundle all signals into two external ports: a *request* port, encompassing signals from the AR, AW, and W AXI channels, and a *response* port, which groups signals from the R and B AXI channels. Master modules initiate AXI transactions by issuing requests and awaiting the corresponding responses. Slave modules receive and process these requests and generate responses. For a more comprehensive understanding of the AXI protocol's operation and its associated signals, we recommend referring to the AMBA AXI protocol specification ([48]). Below, we provide a detailed breakdown of how each interface is internally structured within the IOMMU.

4.6.1 Programming Interface

The programming interface serves as a slave interface used by RISC-V harts for programming and monitoring the memory-mapped registers of the IOMMU. As mentioned in section 4.5.1, the register map exclusively supports 32-bit accesses. For this reason, we integrated a conversion module that translates 64-bit AXI accesses into two consecutive 32-bit accesses and bypasses native 32-bit transactions. Sub-

sequently, a secondary module converts the interface AXI protocol to the register-based protocol used by the register map module. Therefore, the interface shown in Figure 4.25 on the right connects directly to the left interface of Figure 4.19.

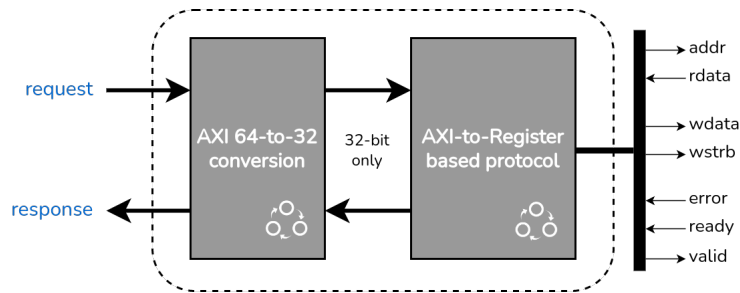


Figure 4.25: Programming interface. The register-based interface on the right connects directly to the interface on the left side of Figure 4.19.

4.6.2 Data Structures Interface

Within the IOMMU, specific internal modules carry out implicit memory accesses as part of the address translation process to fetch memory-based data structures (e.g., device contexts, process contexts and page tables). Other modules require memory access for IOMMU-related management tasks (e.g., fetch commands, report faults and send interrupts). The data structures interface is a master interface used by these modules to access memory. As depicted in Figure 4.26, we use internal arbitration and routing logic to manage the utilization of this single interface by all five associated modules. AXI requests are arbitrated with a fair round-robin policy to provide equal throughput distribution among all master modules. Then, responses are correspondingly routed to the master modules according to their unique AXI IDs, which differ for read and write requests. Notably, for write requests, we incorporated an internal FIFO to propagate the arbitrated IDs and connect the appropriated source module to the W channel of the interface.

4.6.3 Translation Request and Completion Interfaces

All DMA requests are received by the IOMMU and forwarded to main memory through these two interfaces. The IOMMU processes DMA requests sequentially, i.e., an incoming request is stalled until the IOMMU completes the processing of the current one. To facilitate its processing, the IOMMU directly accesses the required data from the AXI bus of the translation request interface, with the *AxVALID* signal serving as an initiation flag for request processing. Besides showing the internal logic embedded between the request and completion interfaces, Figure 4.27 presents all AXI signals used by the IOMMU and their

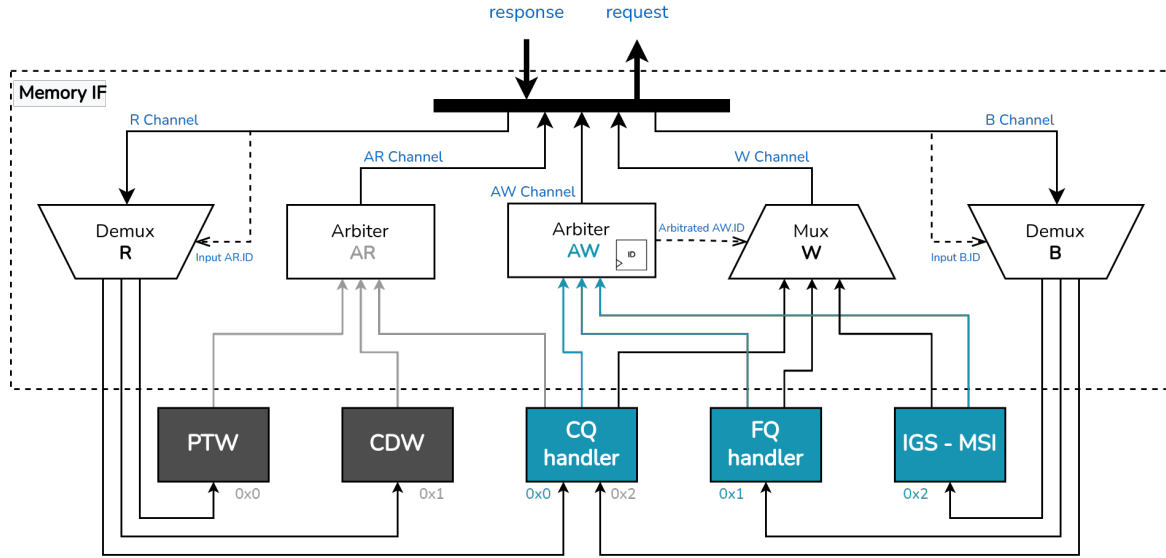


Figure 4.26: Data structures interface. The CDW and PTW modules only issue read requests, the FQ handler and the MSI interrupt generation module only perform write requests, and the CQ handler module initiates both read and write requests.

meaning within the scope of the RISC-V IOMMU specification. We extended the AXI bus of the translation request interface to incorporate IOMMU-related signals compliant with the latest version of the AXI specification (see Chapter A14 of [48]). Although this extension was designed for ARM-based systems, it can be adapted for use with other architectures, such as RISC-V. Furthermore, we included an optional block to check whether transactions adhere to the address boundary limits of the AXI protocol⁹. The *IncIBC* parameter determines the inclusion of this module.

The translation logic block is placed between the translation request interface (or the boundary check block, if included) and the completion interface. Its primary role is to execute address translation following the flowchart depicted in Figure 4.13. The translation logic block intercepts the *AxVALID* and *AxADDR* signals and replaces them using an auxiliary bus. In contrast, the R, W, and B channels remain unchanged and are directly linked to the completion interface. Upon successful request, the translated address is inserted into the *AxADDR* signal of the auxiliary bus, which is then routed to the completion interface. Consequently, the request, now equipped with the translated address, is forwarded to main memory and subsequent responses are delivered directly to the requesting DMA device via the response channels. In the occurrence of an error, whether originating from the boundary check block or the translation logic block, the auxiliary bus (and consequently, the request interface) is rerouted to a dummy slave module

⁹The AXI specification states that bursts must not cross 4-kB address boundaries, thus ensuring they will never cross from one physical slave to another.

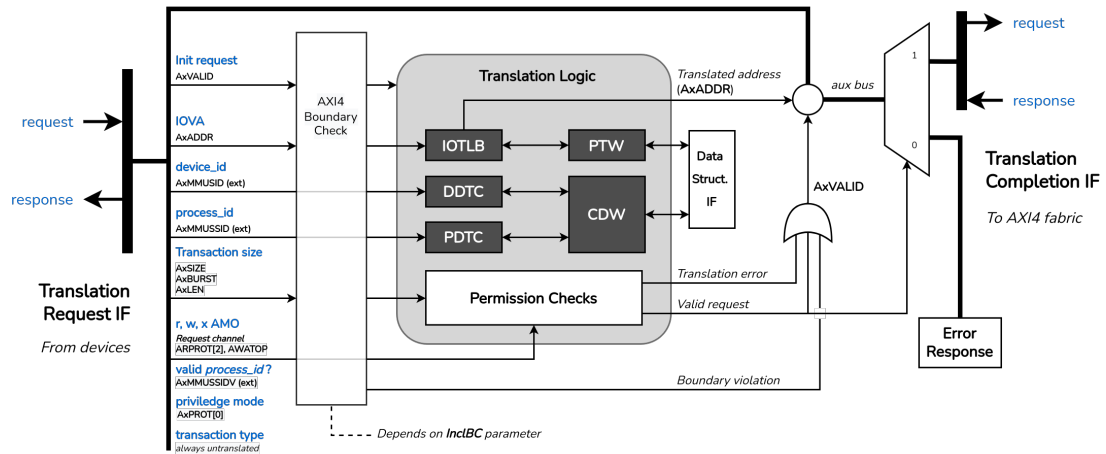


Figure 4.27: Translation request and completion interfaces. Depending on whether the transaction results in success or error, the translation request interface is connected to the completion interface or an error responder.

that consistently responds with errors. In both scenarios, i.e., success and error, the *AxVALID* signal of the auxiliary bus is set accordingly to continue the transaction.

Chapter 5

RISC-V IOMMU Implementation

In the previous chapter, we described all the microarchitectural components of our IOMMU. Nevertheless, certain aspects are left unspecified during the design phase as they concern exclusively the implementation process. In this chapter, we provide overall insights regarding the implementation of the IOMMU IP. Additionally, we describe the implementation of the RISC-V IOMMU driver for Bao hypervisor as part of the software components required for a fully virtualized system stack. For a better understanding, we suggest the reader to accompany the following sections of this chapter with the implementation code and artifacts, made available in [49] (IOMMU IP) and [50] (Bao Hypervisor). All code listings specify the location of the files they refer to from the top directory of the corresponding GitHub repository.

5.1 Key Implementation Details

The implementation of the IOMMU IP was carried out using the SystemVerilog HDL¹⁰. In this sense, we implemented each microarchitectural component within the IOMMU as a SystemVerilog module. We integrated all hardware modules and interfaces according to the microarchitecture presented in Figure 4.12. The top module of the IOMMU is named **riscv_iommu**. It consists of two major wrapper modules: the translation logic wrapper (**rv_iommu_translation_wrapper**), corresponding to the translation logic block described in section 4.4, and the software interface wrapper (**rv_iommu_sw_if_wrapper**), representing the software interface block described in section 4.5. These wrappers instantiate their corresponding modules, as described in the previous chapter. We also implemented wrapper modules for two external interfaces: one for the *programming interface* to encapsulate the conversion logic presented in Figure 4.25 and another for the *data structures interface*, containing the arbitration and routing logic shown in Figure 4.26. These modules are instantiated within the *riscv_iommu* top module, as they are

¹⁰Both the CVA6 core and the target CVA6-based SoC are implemented in SystemVerilog

directly connected to the external AXI ports of the IOMMU.

5.1.1 Parameterizable Implementation

In Chapter 4, we defined a set of design parameters and described their behavioral and microarchitectural effect in the different modules or internal structures of the IOMMU IP. We implemented this parameterized design using SystemVerilog *module parameters* and *generate* statements. This feature enables conditional synthesis and implementation of hardware logic within the IP according to parameter values. Thus, a design parameter can determine the inclusion of hardware structures to implement an optional feature of the architecture, configure microarchitectural properties of these hardware structures, or both. We illustrate this implementation approach with code snippets concerning the WSI generation subsystem. Listing 5.1 shows the instantiation of the **rv_iommu_wsi_ig** module within the software interface wrapper. The module is instantiated if the *IGS* parameter is configured with *WSI_ONLY* or *WSI_MSI*; otherwise, all output interrupt wires (*wsi_wires_o*) are hardwired to zero to be further optimized by the synthesizer.

```
1 (...)  
2 // WSI Interrupt Generation  
3 generate  
4 if ((IGS == rv_iommu::WSI_MSI) || (IGS == rv_iommu::WSI_ONLY)) begin : gen_wsi_ig_support  
5  
6     rv_iommu_wsi_ig #(  
7         .N_INT_VEC ( N_INT_VEC ),  
8         .N_INT_SRCS ( 3 )  
9     ) i_rv_iommu_wsi_ig (  
10         // fctl.wsi  
11         .wsi_en_i ( reg2hw.fctl.wsi.q ),  
12         // ipsr  
13         .intp_i ( {reg2hw.ipsr.pmip.q, reg2hw.ipsr.fip.q, reg2hw.ipsr.cip.q} ),  
14         // icvec  
15         .intv_i ( intv ),  
16         // interrupt wires  
17         .wsi_wires_o ( wsi_wires_o )  
18     );  
19 end : gen_wsi_ig_support  
20  
21 // Hardwire WSI wires to 0  
22 else begin : gen_wsi_support_disabled  
23     assign wsi_wires_o = '0;  
24 end : gen_wsi_support_disabled  
25 endgenerate
```

Listing 5.1: Instantiation of the WSI generation module within the IOMMU software interface wrapper
(*rtl/software_interface/wrapper/rv_iommu_sw_if_wrapper.sv*).

Additionally, the N_INT_VEC parameter determines the number of interrupt vectors supported by the IOMMU. Concerning the WSI generation mechanism, each interrupt vector is represented by an interrupt wire. Thereby, the length of the *ws_i_wires_o* array is based on this parameter (see Listing 5.2, line 4). Interrupt wires are connected to the *intp_i* signal, which holds the values of the interrupt pending bits within the *ipsr* register. The *intv_i* signal holds the values of the interrupt vectors in the *icvec* register and is used as an index to the *ws_i_wires_o* array.

```

1 module rv_iommu_wsi_ig #(
2     (...)
3     // interrupt wires
4     output logic [(N_INT_VEC-1):0] wsi_wires_o
5 );
6     always_comb begin : wsi_support
7         wsi_wires_o = '0;
8         // If WSI generation supported and enabled
9         if (wsi_en_i) begin
10             for (int unsigned i = 0; i < N_INT_SRCS; i++) begin
11                 wsi_wires_o[intv_i[i]] = intp_i[i];
12             end
13         end
14     end
15 endmodule

```

Listing 5.2: Internal logic of the IOMMU WSI generation module
(*rtl/software_interface/rv_iommu_wsi_ig.sv*). The *for* cycle is used to perform connections for each interrupt source.

Furthermore, regarding the interrupt generation subsystem, the length of each field within the *icvec* register is defined by $\log_2(N_INT_VEC)$. Listing 5.3 depicts the conditional instantiation of the *icvec* register fields within the register map module. We use the $LOG2_INTVEC$ parameter to define the size of each field. Two conditions are evaluated: In line 6, it is checked whether the IOMMU supports more than one interrupt vector; otherwise, all register fields are hardwired to zero. In line 16, it is checked whether the IOMMU includes the HPM, as there is no need to include the *pmiv* field in the opposite case.

```

1 module rv_iommu_regmap #(
2     (...)
3     parameter int unsigned LOG2_INTVEC = $clog2(N_INT_VEC)
4     // R[icvec]
5     generate

```

```

6   if (LOG2_INTVEC > 0) begin : gen_icvec
7       // F[civ]
8       rv_iommu_field #(
9           .DATA_WIDTH(LOG2_INTVEC),
10          .SwAccess(SwAccessRW),
11          .RESVAL('0)
12      ) u_icvec_civ (...);
13      // F[fiv]
14      rv_iommu_field #(...) u_icvec_fiv (...);
15      // F[pmiv]
16      if (N_IOHPMCTR > 0) begin : gen_hpm
17          rv_iommu_field #(...) u_icvec_pmiv (...);
18      end : gen_hpm
19  end
20  (...)
21 endmodule

```

Listing 5.3: Instantiation of the *icvec* register within the IOMMU register map

(*rtl/software_interface/regmap/rv_iommu_regmap.sv*). The length of each field is defined by

$$LOG2_INTVEC = \log_2(N_INT_VEC), \text{ as shown in lines 3 and 9.}$$

5.1.2 The Finite State Machine Model

The following modules within the IOMMU were conceived as FSMs: the PTW, the CDW, the CQ and FQ handler modules, and the MSI generation module. In this section, we describe the adopted SystemVerilog FSM model. To do so, we illustrate the implementation of the second-level FSM used by the FQ handler module to write fault records into the FQ (see Figure 4.22). Throughout this section, we present code snippets from the **rv_iommu_fq_handler** module, mainly focusing on the *WRITE* state of the first-level FSM. Overall, the adopted FSM model encompasses two logic blocks: (i) a combinational logic block to determine the behavior of each state by defining output values and next-state values for registers, and (ii) a sequential logic block that specifies the reset state and attributes register next-state values defined by the combinational block.

Listing 5.4 shows the initial part of the FSM’s combinational block, delimited with the *always_comb* keyword in line 8. The first assignments are default values for output signals and FSM registers. We conventionally name output signals with the “_o” suffix, current-state register values with the “_q” suffix, and next-state register values with the “_n” suffix. Note that the default value for next-state register signals is, generally, the current-state value. The state of the FSM is defined by two key signal pairs: *state* for the first-level FSM and *wr_state* for the second-level FSM. The *case* statement in line 17 is used to parse the

current state of the first-level FSM according to the encodings defined in line 4. Within the *case* statement, combinational logic is defined for each state to rule the operation of the FSM.

```

1 module rv_iommu_fq_handler #(
2     (...))
3     // First-level FSM States
4     enum logic [1:0] {IDLE, WRITE, ERROR} state_q, state_n;
5     // Second-level FSM States
6     enum logic [1:0] {AW_REQ, W_DATA, B_RESP} wr_state_q, wr_state_n;
7     (...)
8     always_comb begin // Combinational logic
9         // Default values for outputs
10        (other outputs...)
11        error_wen_o = 1'b0;
12        fq_ip_o     = 1'b0;
13        // Default values for registers
14        state_n     = state_q;
15        wr_state_n  = wr_state_q;
16        (other registers...)
17        case (state_q) // First-level FSM
18            (first-level FSM states...)

```

Listing 5.4: Finite state machine of the FQ handler module, snippet n°1: state encodings and default values (*rtl/software_interface/rv_iommu_fq_handler.sv*).

We focus mainly on the *WRITE* state, shown in Listing 5.5. Another *case* statement is used in line 2 to parse the current state of the second-level FSM based on the encodings of Listing 5.4, line 6. Each state defines output and next-state values, including the *wr_state_n* control signal. Note that the state diagram of Figure 4.22 (on the right) summarizes the flow of this FSM. In the final state (*B_RESP*), the second-level FSM terminates its operation, defining the next state of the first-level FSM, *IDLE* or *ERROR*. We defined a default case (in line 33) to remain in a known state upon FSM desync.

```

1     WRITE: begin // Write fault record to FQ
2         case (wr_state_q) // Second-level FSM
3
4             AW_REQ: begin // Send request to AW Channel
5                 mem_req_o.aw_valid = 1'b1;
6                 if (mem_resp_i.aw_ready) begin
7                     wr_state_n = W_DATA;
8                 end
9             end
10
11            W_DATA: begin // Send data through W channel
12                (fault record write logic...)
13                if (&wr_cnt_q)
14                    wr_state_n = B_RESP;
15            end

```

```

16
17         B_RESP: begin // Check B response code
18             if (mem_resp_i.b_valid) begin
19                 mem_req_o.b_ready = 1'b1;
20                 fq_ip_o          = fq_ie_i; // set ipsr.fip if fqcsr.fie = 1
21                 if (mem_resp_i.b.resp != axi_pkg::RESP_OKAY) begin
22                     state_n      = ERROR; // AXI error
23                     fq_mf_o       = 1'b1;
24                     error_wen_o  = 1'b1;
25                 end else begin
26                     // Successful write. Increment fqt
27                     fq_tail_o = (fq_tail_i + 1) & ~({32{1'b1}} << (fq_size_i+1));
28                     state_n    = IDLE;
29                 end
30             end
31         end
32         // Go to IDLE upon desync
33         default: state_n = IDLE;
34     endcase
35 end

```

Listing 5.5: Finite state machine of the FQ handler module, snippet n°2: *WRITE* state.

The sequential logic block of the FSM under consideration is presented in Listing 5.6, delimited by the *always_ff* keyword. As previously mentioned, the reset state of the entire FSM is defined by this block, starting in line 3. Due to space constraints, we only show the *state* and *wr_state* signal pairs. The first-level FSM starts in the *IDLE* state upon reset, whereas the second-level FSM starts in the *AW_REQ*. From line 7 onwards, next-state values ("*_n*") are assigned to the current-state signals ("*_q*").

```

1     always_ff @(posedge clk_i or negedge rst_ni) begin // Sequential logic
2         if (~rst_ni) begin // Reset values
3             state_q      <= IDLE;
4             wr_state_q   <= AW_REQ;
5             (...)
6         end else begin // Next values
7             state_q      <= state_n;
8             wr_state_q   <= wr_state_n;
9             (...)
10    endmodule

```

Listing 5.6: Finite state machine of the FQ handler module, snippet n°3: sequential logic.

5.1.3 IOMMU Packages

In addition to the IOMMU hardware modules and wrappers, we developed a SystemVerilog package to group relevant IOMMU architectural data and required functions, making the implementation code more

readable and organized. The **rv_iommu** package, located in *packages/rv_iommu_pkg.sv*, declares encodings and data structure molds (*typedefs*) according to the architecture specification. Typedefs facilitate granular access to complex data structures by casting raw arrays of significant dimensions into hierarchical structures. For example, by using this approach, we can access the DC as an opaque 64-byte array or as eight arrays of 64 bits (considering MSI translation enabled), and address each DC field or sub-field individually. Listing 5.7 shows an example of the use of the package. This code snippet belongs to the CQ handler module, and we mainly focus on the processing of the *IODIR* cache invalidation command. All IOMMU commands are fetched as raw arrangements of 128 bits and then casted to a generic command structure (*rv_iommu::cq_entry_t*) to parse its opcode and *func3* field. Then, we cast it to the corresponding command type (*rv_iommu::cq_iodirinv_t* in this case) for easier processing. The encodings and structures for all IOMMU commands are defined in SystemVerilog code within the *rv_iommu* package.

```

1 module rv_iommu_cq_handler #(
2     (...)
3     // Cast read bus to receive CQ entries from memory
4     rv_iommu::cq_entry_t      cq_entry;
5     assign cq_entry            = rv_iommu::cq_entry_t'(cmd_q);
6     rv_iommu::cq_iodirinv_t   cmd_iodirinv;
7     assign cmd_iodirinv       = rv_iommu::cq_iodirinv_t'(cmd_q);
8     (...)
9     case (cq_entry.opcode)
10         flush_dv_o   = cmd_iodirinv.dv;
11         flush_did_o  = cmd_iodirinv.did;
12         rv_iommu::IODIR: begin // IODIR
13             // A command is determined to be illegal if a reserved bit is set to 1
14             // The PID operand is reserved for IODIR.INVALID_DDT
15             // The DV operand must be 1 for IODIR.INVALID_PDT
16             if ((|cmd_iodirinv.reserved_1) || (|cmd_iodirinv.reserved_2) ||
17                 (|cmd_iodirinv.reserved_3) || (|cmd_iodirinv.reserved_4) ||
18                 (cmd_iodirinv.func3 == rv_iommu::DDT && |cmd_iodirinv.pid) ||
19                 (cmd_iodirinv.func3 == rv_iommu::PDT && !cmd_iodirinv.dv) ) begin
20
21                 cmd_ill_o   = 1'b1;
22                 error_wen_o = 1'b1;
23                 state_n     = ERROR;
24             end
25             // Check func3 to determine if command is INVALID_DDT or INVALID_PDT
26             else if (cmd_iodirinv.func3 == rv_iommu::DDT) begin
27                 flush_ddtc_o   = 1'b1;
28                 flush_pdtc_o   = 1'b1;
29             end else if (cmd_iodirinv.func3 == rv_iommu::PDT) begin
30                 flush_pdtc_o   = 1'b1;
31                 flush_pv_o     = 1'b1;
32                 flush_pid_o    = cmd_iodirinv.pid;

```

```

33         end
34     end
35     (...)
36 endmodule

```

Listing 5.7: Processing of the *IODIR* command (*rtl/software_interface/rv_iommu_cq_handler.sv*).

Encodings for operands and command typedef structures are defined in the *rv_iommu* package.

We developed two additional SystemVerilog packages related to the IOMMU register map: ***rv_iommu_reg_pkg*** and ***rv_iommu_field_pkg***. Although the purpose of these packages is similar to that of the *rv_iommu* package, we decided to keep them separated due to the difference in size between the register map and the remaining data structures concerning the IOMMU architecture.

5.1.4 Third-party Resources

Repository	Module	Function
AXI¹	<i>axi_demux</i>	Implement the demux block of Figure 4.27
	<i>axi_err_slv</i>	Implement the error response module of Figure 4.27
Common Cells²	<i>stream_arbiter</i>	Implement the AW and AR arbiters within the data structures interface (Figure 4.26)
	<i>stream_mux</i>	Implement the W mux of Figure 4.26
	<i>stream_demux</i>	Implement the R and B demuxes of Figure 4.26
	<i>fifo_v3</i>	Implement the fall-through FIFO used in the FQ handler module (see Section 4.5.3), and the <i>AWID</i> FIFO of Figure 4.26
AXI to APB bridge³	<i>axi2apb_64_32</i>	Perform protocol conversion between 64-bit AXI4 and 32-bit APB within the programming interface (see Section 4.6.1)
Register Interface⁴	<i>apb_to_reg</i>	Perform protocol conversion between 32-bit APB and 32-bit register-based interface within the programming interface

⁽¹⁾ <https://github.com/pulp-platform/axi>.

⁽²⁾ https://github.com/pulp-platform/common_cells.

⁽³⁾ <https://github.com/pulp-platform/axi2apb>.

⁽⁴⁾ https://github.com/pulp-platform/register_interface.

Table 5.11: Third-party standalone modules used to implement the IOMMU IP.

To accelerate the implementation process, we leveraged open-source SystemVerilog standalone modules to implement certain IOMMU internal functions. Furthermore, we integrated into the IP repository additional packages required by these modules to operate. All third-party modules and packages were

developed by the PULP platform team¹¹ and belong to different repositories according to their purpose. Table 5.11 outlines all third-party modules used out-of-the-box in our implementation. The IOMMU IP repository also includes the dependencies of those modules.

5.2 RISC-V IOMMU Driver for Bao Hypervisor

The developed IOMMU IP is intended to be integrated and evaluated within a virtualized system. The target CVA6-based platform features hardware support for virtualization at the core level, provided by the RISC-V hypervisor extension [5]. Virtualization support at the system level requires additional hardware modules external to the core. In a RISC-V platform, these modules are mainly the IOMMU and the RISC-V AIA [46]. As for the software requirements for virtualization, we chose the Bao hypervisor to run on this platform. Consequently, the second goal of this dissertation is to add RISC-V IOMMU support for Bao, by implementing a software driver. Concerning the RISC-V architecture, Bao currently assigns DMA devices to VMs and passes through their accesses to main memory without any verification, i.e., DMA devices can freely issue memory transfers using physical addresses to the whole address space available within the platform. The RISC-V IOMMU driver for Bao aims to program the IOMMU to constrain the physical address space accessible by DMA devices. On the first iteration, the driver configures the IOMMU to implement the *device pass-through* model (see Section 3.5), providing inter-VM memory protection.

5.2.1 Bao Data Structuring

Before delving into the implementation details of the RISC-V IOMMU driver for Bao, it is helpful to begin by introducing the internal data structure of the hypervisor. Bao organizes its internal data into two distinct categories: those that remain invariant across different target architectures and pertain to the core functionality of the hypervisor, and those that vary depending on the specific target architecture, thereby differing for each supported architecture. Similarly, most of Bao's source code is segregated into two distinct directories: *src/core*, housing architecture-independent code and *src/arch*, containing architecture-dependent code. The class diagram of Figure 5.28 provides a visual representation of several internal data structures within Bao and their associations, highlighting architecture-dependent structures and fields. Due to space constraints, we focus only on those related to the RISC-V IOMMU driver and their parent structures.

¹¹The Parallel Ultra Low Power (PULP) Platform is a joint effort between the ETH Zürich and the University of Bologna to explore new and efficient architectures for ultra-low-power processing. See more at <https://pulp-platform.org/>.

5.2.2 IOMMU Driver

The implementation of the RISC-V IOMMU Bao driver can be found in *src/arch/riscv/inc/arch/iommu.h* and *src/arch/riscv/iommu.c*. The former file serves the simple purpose of defining the **iommu_vm_arch** structure, designed to hold VM and architecture-specific data. In the current iteration of the RISC-V IOMMU driver, this structure remains empty, as indicated at the bottom of the class diagram in Figure 5.28. However, future iterations of the driver may require the addition of fields to this structure. Most of the driver's implementation code resides in the latter file, *iommu.c*. Bao uses three interface functions to facilitate the coupling between the core code and the architecture-dependent IOMMU code: (i) *iommu_arch_init()* performs initial configuration of the IOMMU hardware IP, (ii) *iommu_arch_vm_add_device(...)* assigns a device to a VM and configures the corresponding DC in the DDT, and (iii) *iommu_arch_vm_init(...)* is intended to initialize VM and arch-specific IOMMU data. Implementing the RISC-V IOMMU driver consists in coding these functions, ensuring their compliance with the architecture specification.

```
1 bool iommu_arch_init()
2 {
3     // platform.arch.iommu.base defines whether an IOMMU is present in the platform
4     if(cpu()->id == CPU_MASTER && platform.arch.iommu.base){
5         /* Map programming interface, read and check caps, configure interrupts,
6          map and configure FQ, map and configure DDT, enable IOMMU */
7         rv_iommu_init();
8         return true;
9     }
10    return false;
11 }
```

Listing 5.8: Implementation of the *iommu_arch_init* interface function. Initialization of the IOMMU hardware (*src/arch/riscv/iommu.c*)

In order to accommodate platform-related IOMMU data, we introduced a new field within the input platform structure: the **iommu** structure (accessible via *platform.arch.iommu*), highlighted in dark blue within the Figure 5.28 diagram. Given that this information pertains specifically to the RISC-V architecture, it is incorporated into the **arch_platform** structure. To indicate the presence of RISC-V IOMMU hardware, the user must specify the base address of the memory-mapped programming interface in *platform.arch.iommu.base*. The implementation of the *iommu_arch_init()* interface function can be seen in Listing 5.8. Note that if the base address remains undefined (as indicated in line 4), no IOMMU initialization occurs.

```
1 void rv_iommu_init(void) {
2     (...)
```

```

3 // Allocate IRQ for FQ
4 interrupts_reserve(platform.arch.iommu.fq_irq_id, rv_iommu_fq_irq_handler);
5 interrupts_cpu_enable(platform.arch.iommu.fq_irq_id, true);
6 (...)
7 // Allocate a page of memory (aligned) for the DDT
8 vaddr_t ddt_vaddr = (vaddr_t)mem_alloc_page(
9     NUM_PAGES(sizeof(struct ddt_entry) * DDT_N_ENTRIES), SEC_HYP_GLOBAL, true);
10 // Clear entries
11 memset((void*)ddt_vaddr, 0, sizeof(struct ddt_entry) * DDT_N_ENTRIES);
12 rv_iommu.hw.ddt = (struct ddt_entry*)ddt_vaddr;
13
14 // Configure ddtp with DDT base address and IOMMU mode
15 paddr_t ddt_paddr;
16 mem_translate(&cpu()->as, ddt_vaddr, &ddt_paddr);
17 rv_iommu.hw.reg_ptr->ddtp = (unsigned long long)platform.arch.iommu.mode |
18     ((ddt_paddr >> 2) & RV_IOMMU_DDTP_PPN_MASK);
19 }

```

Listing 5.9: Partial implementation of the *rv_iommu_init* function. Allocation of interrupt line for the FQ and enabling of the IOMMU.

The *rv_iommu_init()* function consolidates all initialization code, particularly addressing the tasks outlined in the comments within lines 5 and 6 of Listing 5.8. For its part, Listing 5.9 shows relevant segments of the *rv_iommu_init()* function that use of the **iommu** structure. Within this function, the driver allocates and maps memory for the FQ and reserves an IRQ line to report translation faults. To accomplish this, as shown in lines 4 and 5, the driver uses the *platform.arch.iommu.fq_irq_id* field to represent the interrupt ID of the FQ and, at the same time, the index of the corresponding interrupt within the platform's PLIC. Furthermore, as mentioned in Chapter 3, the IOMMU is enabled by writing to the *ddtp* memory-mapped register, specifying a value for the *ddtp.iommu_mode* field other than zero (OFF). The user designates this mode within *platform.arch.iommu.mode*, and it is subsequently written to the corresponding register in line 17.

```

1 static bool iommu_vm_arch_add(struct vm *vm, deviceid_t dev_id) {
2     if (dev_id > 0) {
3         if(rv_iommu_alloc_did(dev_id)) { // Check if device was already added
4             paddr_t rootpt;
5             // Translate root PT base address
6             mem_translate(&cpu()->as, (vaddr_t)vm->as.pt.root, &rootpt);
7             // Set DDT entry with root PT base address, VMID and configuration
8             rv_iommu_write_ddt(dev_id, vm, rootpt);
9         }
10        else {
11            INFO("RV IOMMU: Cannot add device %d twice", dev_id);
12            return false;
13        }
14    }
15 }

```

```

14     }
15     else {
16         INFO("RV IOMMU: Invalid device ID: %d", dev_id);
17         return false;
18     }
19     return true;
20 }

```

Listing 5.10: Implementation of the *iommu_vm_arch_add* function. Assignment of a device to a VM, and DC configuration.

Listing 5.10 illustrates the implementation of the *iommu_vm_arch_add(...)* function, used to assign a device to a VM and configure the corresponding DC. In this process, we use a bitmap structure to keep track of which devices the IOMMU has processed. It is important to note that Bao is a static partitioning hypervisor, and its scope is to enforce isolation between VMs. Consequently, one device cannot be shared between two or more VMs. To configure each DC, we need the VMID tag and the base physical address of the second-stage page table. In Bao's data structure, both parameters are part of the **vm** structure, as outlined in Figure 5.28. Specifically, the VMID is stored in the *vm.id* field, while the base address is contained within *vm.as.pt.root*. Besides these two parameters, the *rv_iommu_write_ddt()* function internally applies the same translation configuration for all VMs and devices. Notably, since Bao leverages the Sv39x4 scheme within the core MMU, all DCs are programmed with an identical translation scheme.

Chapter 6

Integration, Validation, and Evaluation

In this chapter, we outline the process of integrating the developed IOMMU IP into the target CVA6-based SoC to be further validated and evaluated. We provide a brief introduction to the target SoC, specifying the modifications done to include the IOMMU and the required DMA modules. Subsequently, we describe the IOMMU validation plan, which was divided into three phases, each one with a particular scope and associated system setup: (i) a custom baremetal framework, (ii) a Linux-based software stack, and (iii) a virtualization-based software stack based on Bao hypervisor. Additionally, we show relevant outcomes obtained during the validation process. Finally, we conduct a design space exploration to understand the impact of the different architectural features (mandatory vs. optional), as well as microarchitectural structures and parameters. In this final endeavor, we provide insights into FPGA hardware costs and added translation latencies for different IOMMU hardware configurations.

6.1 Platform-level Integration

An IOMMU is naturally a system-level hardware module conceived to mediate requests from DMA-capable peripherals attached to the platform. Thus, an IOMMU is positioned outside the core(s), between DMA-capable devices and system memory. We integrated the developed IOMMU IP into a single-core CVA6-based SoC with virtualization support [5, 51], presented in Figure 6.29¹². This SoC can run at 50MHz and features a highly parameterized AXI crossbar to interconnect the CVA6 core, memory hierarchy components, system peripherals, and other modules. All peripherals are instantiated within a parameterized wrapper module named **ariane_peripherals** in such a way that allows to define which peripherals to incorporate into the SoC at design time, using SystemVerilog parameters. With all peripherals included, the crossbar implements fifteen master ports and four slave ports. To integrate the IOMMU

¹²The implementation code of this SoC is available in [52]

IP, we instantiated the *riscv_iommu* module within the *ariane_peripherals* wrapper. Additionally, we added two master AXI ports in this module to route the IOMMU data structures and completion interfaces to the crossbar. In the crossbar configuration, we assigned an address range of one page (4 kiB) for the IOMMU programming interface, starting at the base address 0x50010000.

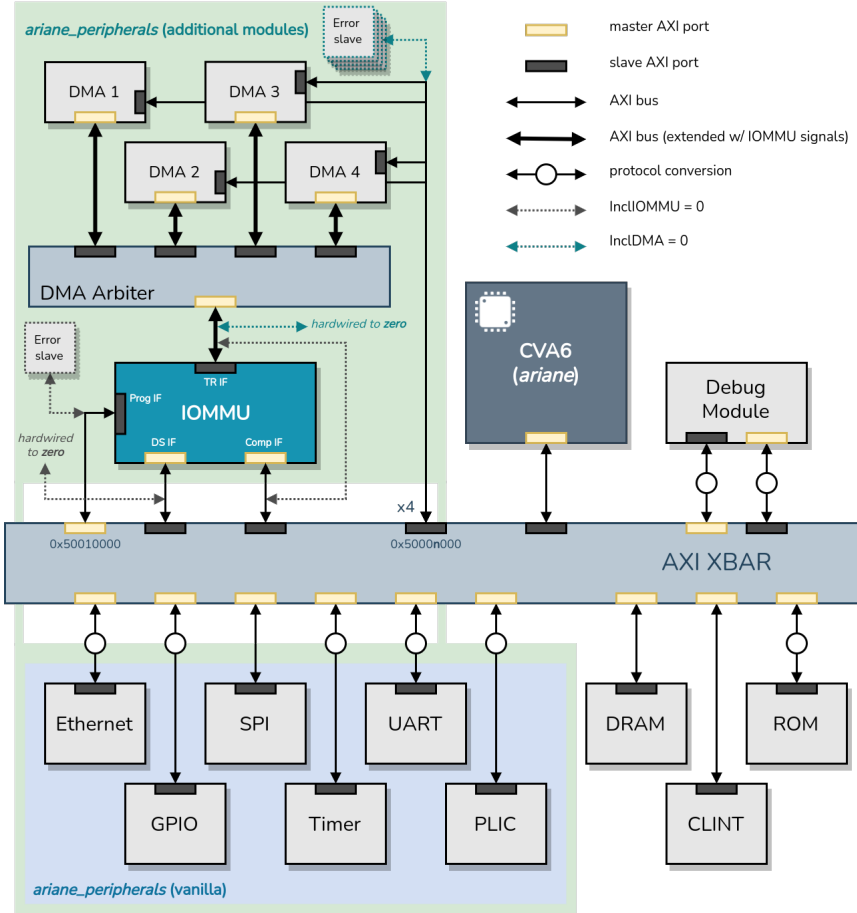


Figure 6.29: Block diagram of the CVA6-based SoC used to validate and evaluate the IOMMU IP. Two design parameters rule the incorporation of the IOMMU IP and DMA modules into the *ariane_peripherals* wrapper.

To validate and evaluate the IOMMU IP, we integrated into the SoC four instances of the *iDMA* SystemVerilog module (**dma_core_wrap** module), developed by the PULP platform team [53]. This module is defined as a *data movement accelerator*, as its operation consists of performing AXI data transfers of configurable size. Each data transfer involves an AXI read from a source address, followed by an AXI write to a destination address. An *iDMA* module has two AXI interfaces: a master port to issue memory transfers and a slave port used by software to configure transfer parameters (e.g., source and destination addresses and transfer size), initiate transfers, and monitor their status. Within the AXI crossbar, we defined four 4-kiB contiguous memory-mapped regions for the configuration interfaces, starting at 0x50000000. Fur-

thermore, we extended the iDMA module with additional features required for our validation and evaluation setup¹³. Firstly, we extended the master interface with IOMMU-related signals from the AXI specification, similarly as we did with the translation request interface of the IOMMU (see Section 4.6.3). Secondly, we included a design parameter to define the device ID of each iDMA instance. Finally, we integrated an interrupt pending register with two bits: *ipsr.rip* and *ipsr.wip*, to signal the completion of the read phase and the write phase of a transfer, respectively.

Since the IOMMU has a single translation request slave port, we developed a DMA arbiter module (***dma_arb***) to mediate and route requests issued by multiple DMA-capable devices, even if they are generated simultaneously. The internal logic of this arbiter is similar to that of the data structures interface of the IOMMU, illustrated in Figure 4.26, using stream arbiters to select requests and mux/demux logic to route back responses. Thus, the *dma_arb* module implements a fair round-robin arbitration policy to select the iDMA device to be routed to the IOMMU translation request interface. Notably, these three hardware layers (iDMA devices, DMA arbiter, and IOMMU IP) are instantiated within the *ariane_peripherals* wrapper module. We use SystemVerilog generate blocks and two parameters to make the incorporation of the IOMMU IP and DMA devices optional independently. Table 6.12 summarizes the effect of all possible combinations for the *InclDMA* and *InclIOMMU* parameters, also illustrated in the SoC diagram of Figure 6.29.

6.2 Functional Validation

Instead of individually testing the internal modules of the IOMMU and the complete IP as an isolated component, we opted for functionally validating the operation of the IOMMU IP integrated into the CVA6-based SoC. For that purpose, we use three different software setups to assess the IOMMU within the platform, using the iDMA modules to issue memory transfers. The validation setup includes (i) a custom baremetal framework (based on the one used in [30]), (ii) a Linux-based software stack (using the RISC-V IOMMU Task Group driver), and (iii) a virtualization-based software stack based on Bao hypervisor.

As for the development tools, all RTL code was synthesized using Vivado v2022.2, and the implementation was carried out on a Genesys2 FPGA board. The software stack comprises OpenSBI v1.0, Linux v6.3.0-rc7, and Bao hypervisor v1.0. OpenSBI and Linux were compiled using the RISC-V GNU toolchain: GCC v12.2.0 for unknown targets and GCC v9.4.0 for Linux targets, respectively. Bao was compiled using SiFive’s GCC v10.2.0. Additional details concerning used tools and versions can be found in the

¹³The implementation code of the iDMA module with modifications is available in [54]

{InclDMA, InclIOMMU}	Description
{0, 0}	No additional module is instantiated. All VALID/READY signals of crossbar slave ports reserved for the IOMMU IP are hardwired to zero. The master ports of the crossbar used by the IOMMU and iDMA programming interfaces are connected each to an <i>axi_err_slv</i> module.
{0, 1} ¹	Only the <i>riscv_iommu</i> module is instantiated. All VALID/READY signals of the IOMMU translation request bus are hardwired to zero. The iDMA programming interfaces are replaced with <i>axi_err_slv</i> modules.
{1, 0}	The <i>dma_arb</i> and four <i>dma_core_wrap</i> modules are instantiated and attached directly to the system crossbar, by connecting the master port of the DMA arbiter to the slave port reserved for the IOMMU completion interface. The IOMMU programming interface is replaced with an <i>axi_err_slv</i> module, and the VALID/READY signals of the data structures interface bus are hardwired to zero.
{1, 1}	All modules are instantiated

⁽¹⁾ Including an IOMMU without DMA devices doesn't offer significant utility. Nevertheless, we provide this option for conventional reasons.

Table 6.12: Effect of the *InclDMA* and *InclIOMMU* parameters in the *ariane_peripherals* wrapper module.

corresponding implementation repositories.

6.2.1 Custom Test Framework

The first level of the validation plan involves granular testing of the architectural features supported by the developed IOMMU IP. For this purpose, we modified the baremetal test framework reported in [30], developed initially to validate the RISC-V hypervisor extension support in the Rocket Chip. We implemented the functions outlined in Listing 6.11, each representing a test addressing one or more IOMMU architectural features¹⁴. The *TEST_REGISTER* macro includes a test in the list of functions to be executed.

```

1 // IOMMU Tests
2 TEST_REGISTER(hpm); // event counters, cycle counter, HPM interrupts (WSI and MSI)
3 TEST_REGISTER(msi_generation); // CQ and FQ interrupt via MSI
4 TEST_REGISTER(iofence); // IOFENCE.C with WSI=1 and AV=1
5 TEST_REGISTER(wsi_generation); // FQ interrupt via WSI
6 TEST_REGISTER(iotinval); // IOTINVAL.VMA and IOTINVAL.GVMA
7 TEST_REGISTER(two_stage_translation); // two-stage trans., msi trans, IODIR.INVALID_DDT
8 TEST_REGISTER(second_stage_only); // 2nd-stage trans. only, msi trans., IODIR.INVALID_DDT
9 TEST_REGISTER(both_stages_bare); // both translation stages in bare mode
10 TEST_REGISTER(iommu_bare); // IOMMU in bare mode
11 TEST_REGISTER(iommu_off); // IOMMU off (all transactions disallowed)

```

¹⁴The implementation code of this test framework is available in [55]

```

12 // iDMA-only Tests (no IOMMU in the platform)
13 TEST_REGISTER(idma_only_multiple_beats); // multi-beat AXI transfers through iDMAs
14 TEST_REGISTER(idma_only); // // single-beat AXI transfers through iDMAs

```

Listing 6.11: Test battery implemented for the baremetal framework. Each test addresses one or more IOMMU architectural features.

Most of these tests follow the same procedure: (1) Configure the IOMMU according to the feature(s) the test validates; (2) configure and initialize one or more DMA transfers; and (3) corroborate the expected results. Furthermore, each test produces results in a boolean form, i.e., reports success or failure. Instead of compiling the framework and running the binary file directly in the FPGA platform, we first leveraged the simulated version of the SoC to run the application. The platform is simulated using the *verilator* SystemVerilog simulator and accepts a binary file to be executed. Additionally, using this simulator, it is possible to generate a trace file with waveforms representing internal signals, which is very useful for locating hardware bugs. In this sense, after several iterations in the simulated SoC, we ran the application in the physical FPGA platform using the synthesized bitstream, completing successful execution with all tests passed. In order to illustrate this process, Listing 6.12 shows the result of the *two_stage_translation* and *msi_generation* tests execution.

```

1 RISC-V Input/Output Memory Management Unit Tests
2 (initial configuration...)
3 (other tests...)
4 two_stage_translation
5 VERBOSE: IOMMU 1LVL mode | iohgatp: Sv39x4 | iosatp: Sv39 | msiptp: Flat
6 Writing IODIR.INVALID_DDT to CQ
7     Two-stage translation: First transfer matches (4kiB pages) PASSED
8     Two-stage translation: Second transfer matches (2MiB superpages) PASSED
9     Two-stage translation: Third transfer matches (1GiB superpages) PASSED
10    Two-stage translation: Fourth transfer matches (MSI translation) PASSED
11 (other tests...)
12 msi_generation
13 VERBOSE: IOMMU 1LVL mode | iohgatp: Sv39x4 | iosatp: Sv39 | msiptp: Flat
14 VERBOSE: CQ interrupt vector masked
15     cqcsr.cmd_ill was set PASSED
16     ipsr.cip and ipsr.fip were set PASSED
17     MSI data corresponding to FQ interrupt vector matches PASSED
18 VERBOSE: CQ interrupt vector unmasked
19     MSI data corresponding to CQ interrupt vector matches after clearing mask PASSED
20 (other tests...)
21 end

```

Listing 6.12: Execution of the *two_stage_translation* and *msi_generation* tests in the physical FPGA platform, using the synthesized bitstream.

6.2.2 System-level Validation

Once the architectural features of the IOMMU IP were functionally validated using the baremetal test framework, we submitted the IP to more realistic and comprehensive software setups: Linux and Bao hypervisor. The scope of this phase was to validate the operation of the IOMMU within more complex software systems, corroborating that it perfectly fitted without causing interference or conflicts. Figure 6.30 illustrates the software stack used for both validation setups, serving as a reference for the following two sections.

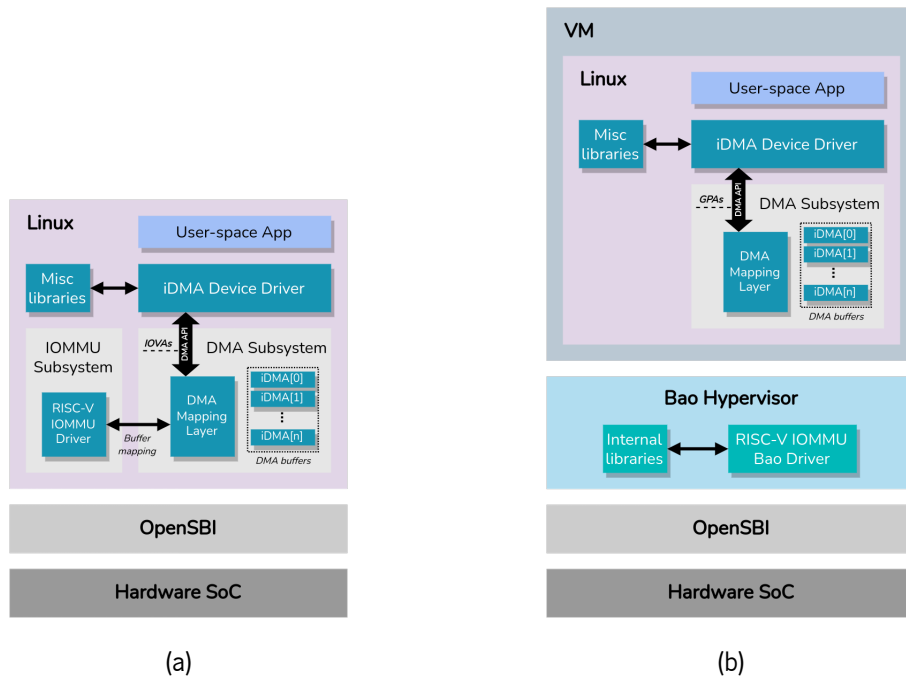


Figure 6.30: Software stack used for system-level validation:

(a) Linux-based setup; (b) virtualization-based setup.

a) Linux-based Software Stack

During the development of the architecture specification, the RISC-V IOMMU Task Group concurrently implemented a proof-of-concept software ecosystem to simulate the RISC-V IOMMU. Among the elements integrating the ecosystem, there is a compliant device implemented in QEMU to model the IOMMU hardware and a Linux driver to configure a RISC-V IOMMU, both open-sourced. In this sense, we leveraged the Linux driver to develop a custom Linux-based software setup for our IOMMU IP¹⁵. The only additional requirement concerning the IOMMU, was to add a device node for the IOMMU IP in the device tree used to

¹⁵The implementation code for the Linux-based software setup is available in [56]

build the Linux image. Within this entry, we report to Linux important IOMMU platform-related information, e.g., the base address of the memory-mapped programming interface, compatible drivers, and interrupt indexes within the platform’s PLIC.

To configure and control memory transfers, we developed a Linux device driver for the iDMA module. Each iDMA device is detected through the corresponding device tree node, and, within the driver, it is registered in the Linux Kernel as a *misc* device. We use the *DMA Application Programming Interface* (API) to allocate a private buffer for each iDMA device and map these buffers invoking the RISC-V IOMMU driver. This API offers two types of DMA mappings: (i) *streaming* mappings; and (ii) *consistent* mappings. We implemented the driver to be configured to use both types of mappings. Nevertheless, streaming mappings are wiped out after each DMA transfer, flushing internal IOMMU caches. Hence, for evaluation purposes, we generally use consistent DMA mappings configured once at kernel initialization time. Listing 6.13 outlines the boot messages printed by the Linux kernel related to the initial configuration of the IOMMU and two iDMA devices. Note that the IOMMU is configured in *1LVL* mode (value of 2), and DMA mappings (blue messages) are created after this configuration.

```

1 (...)
2 [ 11.842667] riscv,iommu 50010000.iommu: CQ length: 128
3 [ 11.850883] riscv,iommu 50010000.iommu: FQ length: 128
4 [ 11.951745] riscv,iommu 50010000.iommu: mode_readback: 2, mode: 2
5 [ 11.960567] riscv,iommu 50010000.iommu: ddt_mode: 2
6 [ 12.763646] il_manolo,idma 50000000.master: Adding to iommu group 0
7 [ 12.774714] il_manolo,idma 50000000.master: domain id 1 S-Stage type 3 attach
8 [ 12.896326] il_manolo,idma 50000000.master: Write Completion IRQ: 16
9 [ 12.944334] iommu: Mapping IOVA 3ffffff000 to SPA 82b90000
10 [ 12.955280] iommu: Mapping IOVA 3ffffffe000 to SPA 82b93000
11 [ 12.963844] il_manolo,idma 50000000.master: Registering iDMA0 device
12 [ 12.979748] il_manolo,idma 50001000.master: Adding to iommu group 1
13 [ 12.991129] il_manolo,idma 50001000.master: domain id 2 S-Stage type 3 attach
14 [ 13.090959] il_manolo,idma 50001000.master: Write Completion IRQ: 17
15 [ 13.146734] iommu: Mapping IOVA 3ffffff000 to SPA 82b9e000
16 [ 13.156440] iommu: Mapping IOVA 3ffffffe000 to SPA 82ba1000
17 [ 13.164979] il_manolo,idma 50001000.master: Registering iDMA1 device
18 (...)

```

Listing 6.13: Linux boot messages related to the IOMMU and iDMA modules. Note that identical IOVAs are mapped to different GPAs.

The iDMA driver is invoked by a user-space application developed to issue memory transfers between two 4-kiB user-space buffers. We use the file operations interface provided by the Linux kernel for character device drivers to implement the transfer procedure. The *read* and *write* functions within the device driver copy the contents of a user-space source buffer to the DMA buffers, configure the corresponding iDMA

module, and trigger the DMA transfer. Listing 6.14 presents the output messages of two DMA transfers in the Linux-based setup. The first transfer (lines 1-4) is well configured. Thus, the test string in the source buffer is copied to the destination buffer. Note that the IOVAs used are those configured at initialization (see Listing 6.13). For the second transfer (lines 6-15), we set a module parameter that assigns unmapped addresses to the source and destination registers of the iDMA module. In this case, a page fault occurs, and a new record is written into the FQ by the IOMMU, subsequently generating a wired interrupt. The kernel receives the interrupt and invokes a callback within the IOMMU driver to report fault data (e.g., fault cause code, associated IOVAs, among others).

```

1 # /etc/test_idma.elf 1
2 [ 480.776069] Using src DMA Addr -> 3ffffff000
3 [ 480.783467] Using dst DMA Addr -> 3fffffe000
4 Src: Test string | Dst: Test string
5 # echo 1 > /sys/module/idma/parameters/use_unmapped_addr
6 # /etc/test_idma.elf 1
7 [ 505.983866] Using src DMA Addr -> 3fc4ca1000
8 [ 505.991160] Using dst DMA Addr -> 3fc4ca2000
9 [ 506.000664] riscv,iommu 50010000.iommu: F: 13 | DID: 1
10 [ 506.008351] riscv,iommu 50010000.iommu: iotval: 3fc4ca1000
11 [ 506.016135] riscv,iommu 50010000.iommu: iotval2: 0
12 [ 506.023359] riscv,iommu 50010000.iommu: F: 15 | DID: 1
13 [ 506.031228] riscv,iommu 50010000.iommu: iotval: 3fc4ca1000
14 [ 506.039422] riscv,iommu 50010000.iommu: iotval2: 0
15 Src: Test string | Dst:

```

Listing 6.14: Output messages of two DMA transfers in the Linux-based setup. Fault codes 13 and 15 correspond to load and store page faults, respectively.

To confirm that the DMA requests were correctly received, translated, and internally cached, we wrote additional code in the RISC-V IOMMU driver to program the HPM to count five standard events: Untranslated requests, IOTLB misses, DDTC misses, and first and second-stage page table walks. We executed three consecutive DMA transfers immediately after the initialization of the IOMMU: two using the same iDMA device and the last transfer using a different one. After this experience, the counters within the HPM reported the values shown in Listing 6.15: six untranslated requests (three reads and three writes); four IOTLB misses (the second transfer was translated with the cached mappings); two DDTC misses (the second transfer used the cached DC); four first-stage page table walks (one for each IOTLB miss); and zero second-stage page table walks, as no hypervisor runs below Linux.

```

1 # cat /sys/class/iommu/riscv-iommu@50010000/riscv-iommu/reg_iohpmctr_1
2 0x6      (untranslated requests)
3 # cat /sys/class/iommu/riscv-iommu@50010000/riscv-iommu/reg_iohpmctr_2
4 0x4      (IOTLB misses)

```

```

5 # cat /sys/class/iommu/riscv-iommu@50010000/riscv-iommu/reg_iohpmctr_3
6 0x2      (DDTC misses)
7 # cat /sys/class/iommu/riscv-iommu@50010000/riscv-iommu/reg_iohpmctr_4
8 0x4      (first-stage PTW)
9 # cat /sys/class/iommu/riscv-iommu@50010000/riscv-iommu/reg_iohpmctr_5
10 0x0      (second-stage PTW)

```

Listing 6.15: State of five HPM event counters after three DMA transfers in the Linux-based setup. Each counter is programmed by the IOMMU driver at initialization time to count a particular standard event.

b) Virtualized Software Stack with Bao

After validating the operation of the developed IOMMU IP in Linux, we inserted a virtualization layer into the software stack: Bao hypervisor. The aim of this last phase in the validation plan was primarily to test the RISC-V IOMMU driver developed for Bao and the IP support for second-stage translation within a virtualized environment. It is important to remark that it is only possible to run up to one VM on top of Bao in the current version of the target platform. Bao is a static-partitioning hypervisor, so all physical resources (including physical cores) are distributed among the VMs before execution. As this platform is single-core, only one VM can be run on top of Bao. Hence, we configured Bao with a single VM running the same image used for the Linux-based setup. The DMA API and its correlation with the Linux IOMMU subsystem provide an advantageous abstraction layer that simplifies the implementation of DMA device drivers. For this reason, it is unnecessary to modify any part of the Linux code when removing the IOMMU hardware. The only difference is that, in this setup, the IOMMU IP is not reported to Linux through the device tree. Only the hypervisor knows that IOMMU hardware is available, reported through the input *platform* structure introduced in Section 5.2.1. In Listing 6.16, we outline relevant fields of the VM configuration used for Bao, defined in the *config* structure. Regarding the data they include, the iDMA configuration fields are similar to the analog device tree nodes. Each entry provides information about base addresses and sizes of memory-mapped regions, interrupts, and the ID of the device for IOMMU processing.

```

1 struct config config = {
2     (...)
3     .vmlist_size = 1,
4     .vmlist = [{
5         .image = { .base_addr = 0x80200000, .load_addr = VM_IMAGE_OFFSET(linux_image),
6                   .size = VM_IMAGE_SIZE(linux_image)},
7         .entry = 0x80200000,
8         .platform = {
9             .cpu_num = 1,
10            (other fields...)
11            .dev_num = 9,

```

```

12     .devs = (struct vm_dev_region[]) {
13         (other peripherals...)
14         // iDMA[0]
15         { .pa = 0x50000000, .va = 0x50000000, .size = 0x00001000,
16           .interrupt_num = 0, .interrupts = (irqid_t[]) {}, .id = 1},
17         // iDMA[1]
18         { .pa = 0x50001000, .va = 0x50001000, .size = 0x00001000,
19           .interrupt_num = 0, .interrupts = (irqid_t[]) {}, .id = 2},
20         (remaining iDMAs...)

```

Listing 6.16: VM configuration used for Bao in the virtualized software setup, defined through the *config* structure introduced in Section 5.2.1.

After booting Bao, the Linux image running on the configured VM invokes the iDMA device driver to allocate a DMA buffer for each iDMA device. However, this time, the RISC-V IOMMU driver is not invoked to map these buffers, as Linux is not aware of the IOMMU hardware present in the platform. Consequently, the DMA buffers are accessed using GPAs, viewed as real physical addresses by Linux. After executing the same user-space application used in the Linux-based software setup, we obtained the output shown in Listing 6.17. As in the previous setup, we carried out two DMA transfers: one was adequately configured, and the other was misconfigured. Bao processes the interrupts generated by the FQ, reporting fault data as shown in lines 9 and 10. Fault cause codes 21 and 23 correspond to guest load page fault and guest store page fault, respectively.

```

1 # /etc/test_idma.elf 1
2 [ 78.800026] Using src DMA Addr -> 82b2b000
3 [ 78.806715] Using dst DMA Addr -> 82b2c000
4 Src: Test String | Dst: Test String
5 # echo 1 > /sys/module/idma/parameters/use_unmapped_addr
6 # /etc/test_idma.elf 1
7 [ 102.712338] Using src DMA Addr -> 3fca625000
8 [ 102.719959] Using dst DMA Addr -> 3fca626000
9 BAO WARNING: RV IOMMU FQ: CAUSE: 21 | DID: 1 | iotval: ca625000 | iotval2: ca625000
10 BAO WARNING: RV IOMMU FQ: CAUSE: 23 | DID: 1 | iotval: ca626000 | iotval2: ca626000
11 Src: Test String | Dst:

```

Listing 6.17: Output messages of two DMA transfers in the virtualization-based setup. Interrupts generated by the FQ are handled by Bao.

6.3 Evaluation

After completing the validation phase, we conducted an empirical evaluation concerning the functional and FPGA resource-utilization impact of the IOMMU IP within a virtualization-capable system. Leverag-

ing the parameterized characteristic of the developed IP, we scaled up this analysis for various hardware configurations based on different combinations of IOMMU architectural features (mandatory vs optional) and microarchitectural structures/parameters. The aim was to determine the influence of configurable IOMMU architectural and microarchitectural elements on hardware costs and translation latencies, providing quantitative results for each domain.

6.3.1 Hardware Resources

Using the feature provided by Vivado to generate module-grained resource utilization reports, we obtained detailed data regarding hardware costs for each synthesized setup. Table 6.13 presents FPGA *Look-Up Table* (LUT) and *Flip-Flop* (FF) utilization for different IOMMU configurations, only considering internal modules affected by design parameters. With this information, we aim to understand how specific IOMMU architectural features and microarchitectural properties affect key internal components regarding hardware costs while providing a preliminary reference and baseline for system designers¹⁶. The baseline IOMMU configuration has four IOTLB entries, four DDTC entries, and four interrupt vectors using only WSI generation support. Subsequent IOMMU configurations incorporate additional hardware logic related to the parameterizable features summarized in Table 4.10. We include the increase in percentage relative to the baseline configuration for the most significant variations (cells highlighted with grey/blue). Note that the first four design configurations after the baseline, primarily affect components from the translation logic hardware block (i.e., the modules at the top of the Table). In contrast, the remaining four design configurations mainly influence the software interface block (regmap and HPM).

Analyzing the contents of Table 6.13, we can infer some points about the effect of design parameters in the implementation. For example, the use of FFs increments proportionally in the IOATC with increased cache/IOTLB entries. In addition, introducing MSI translation support with 8 DDTC entries increases DDTC resource usage by more than 300% as the size of the DC doubles. Nevertheless, overall, this just represents around a 10% hardware increase in the overall logic of the SoC (with IOMMU). Then, introducing PC support adds the PDTC and increases CDW and PTW global resource consumption (FFs + LUTs) by 50% and 200%, respectively. Furthermore, the addition of interrupt vectors, MSI generation support, and the HPM predominantly affect the register map due to the inclusion of extra registers (per feature).

Our analysis indicates that the IOATCs and the PTW generally represent the most resource-consuming modules. However, in the last two configurations, the register map prevails regarding hardware consump-

¹⁶There may be deviations in hardware cost assessment between FPGA and ASIC synthesis. We plan to conduct this study in future work by integrating our IOMMU IP into an evolving version of Shaheen [51].

LUTs / FFs	CDW		PTW		IOTLB		DDTC		PDTC	
Baseline ^(*)	35	195	378	263	411	659	1449	539	-	-
x8 DDTC, x16 IOTLB	84	195	323	263	3528(+758%)	2639 (+300%)	633 (-56%)	1079 (+100%)	-	-
+ MSI translation	86	347 (+78%)	969 (+156%)	1378 (+424%)	2488	2655	7108 (+391%)	2295 (+326%)	-	-
x8 DDTC, x32 IOTLB	90	347	981	1378	7459 (+1715%)	5312 (+706%)	2254	2295	-	-
+ PC support, x8 PDTC	218 (+523%)	421 (+116%)	5750 (+1421%)	1510 (+474%)	6356	5312	1788	2311	217	567
+ MSI IGS	155	420	1058	1365	5858	5312	6598	2311	217	567
x16 int. vectors	184	420	5732	1510	5907	5312	2594	2311	217	567
x8 HPM counters	193	420	5853	1510	5264	5312	1784	2311	217	567
x31 HPM counters	189	420	5808	1511	5396	5344	2997	2311	217	567

LUTs / Flip-Flops	Regmap		HPM		Additional Misc Logic				SoC	
					Intra-IOMMU		External			
Baseline ^(*)	265	295	-	-	3032	1549	9206	7975	90655	62406
x8 DDTC, x16 IOTLB	332	296	-	-	2630	1549	9377	7978	92786 (+2.3%)	64930 (+4%)
+ MSI translation	238	296	-	-	3066	1549	9945	8053	99779 (+10%)	67504 (+8.1%)
x8 DDTC, x32 IOTLB	254	296	-	-	3154	1549	10056	8056	100145 (+11%)	70164 (+12%)
+ PC support, x8 PDTC	300	295	-	-	3235	1550	10184	8059	103927 (+15%)	70956 (+14%)
+ MSI IGS	578 (+118%)	644 (+118%)	-	-	3598	1579	10516	8080	104457 (+15%)	71209 (+14%)
x16 int. vectors	1245 (+370%)	1694 (+474%)	-	-	3760	1597	10441	8119	107959 (+19%)	72461 (+16%)
x8 HPM counters	5376 (+1929%)	3825 (+1197%)	343	102	4115	1598	11625	8110	110649 (+22%)	74686 (+20%)
x31 HPM counters	7813 (+2848%)	5760 (+1853%)	590 (+72%)	135 (+32%)	6241	1598	10298	8119	115428 (+27%)	76696 (+23%)

^(*) The Baseline configuration encompasses x4 DDTC, x4 IOTLB, x4 interrupt vectors and WSI-only IGS.

Table 6.13: Hardware resources used by the target SoC with different IOMMU configurations. We highlight the most affected components for each configuration according to the hardware blocks defined in Figure 4.12: (–) translation modules; (•) software interface.

tion due to the incorporation of multiple counter registers. Overall, the IOMMU configured with all features and enhanced with all microarchitectural structures represents an increase of more than 50% in total SoC resource utilization compared to the vanilla version of the SoC. Nevertheless, we highlight that the target CVA6-based SoC is highly optimized for area [5, 51], and this value may be significantly lower in more complex multi-core SoCs.

6.3.2 Translation Latencies

The main goal of functionally evaluating the IOMMU IP is to measure the additional DMA transfer latency introduced by the address translation process in virtualized systems. To carry out this analysis, we followed an approach similar to that used in the validation plan. We configure the iDMA modules to issue untranslated requests to the IOMMU using two benchmarking setups: (i) a baremetal setup focused on a more precise and tailor-made evaluation; and (ii) a system-level setup aimed for a broader evaluation within sophisticated software systems. Furthermore, to determine the influence of configurable IOMMU architectural and microarchitectural elements in the performance of the IP, we use multiple design configurations for each setup.

a) Baremetal Benchmarking

In this setup, we developed a baremetal application to conduct the benchmarking procedure. This application configures the IOMMU to perform two-stage translation for all devices using a set of 32 guest-virtual-to-supervisor-physical mappings (i.e., page tables). Additionally, it programs the DDT with a different GSCID tag (VMID) for each iDMA, so that each device is seen by the IOMMU as assigned to a different VM. To measure transfer latencies, we leveraged the cycle counters of the CVA6 core, and we used the IOMMU internal HPM to count IOATC misses.

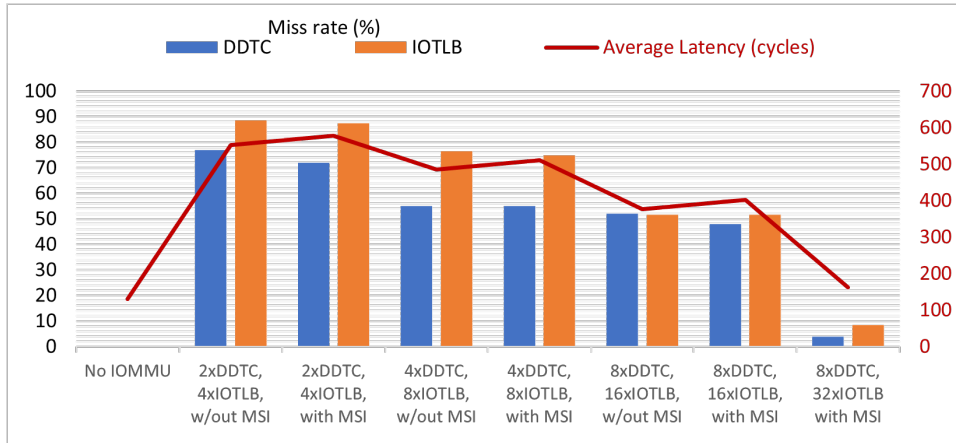


Figure 6.31: Average latency in clock cycles of 1000 DMA transfers in the baremetal setup, using various IOMMU microarchitectural configurations. The most significant latency (~ 550 cycles) is obtained with the third configuration (x2 DDTC entries, x4 IOTLB entries, and MSI translation).

Figure 6.31 presents the average latency, in clock cycles, of 1000 DMA transfers carried out using the IOMMU IP configured with different microarchitectural parameters. We focused on the parameters directly impacting the address translation timing (i.e., IOATC sizes and MSI translation support). We first considered the scenario with the IOMMU disabled, obtaining a base latency of ~ 130 cycles. Then, for naive microarchitectural configurations (e.g., x2 DDTC entries, x4 IOTLB entries, and MSI translation), the DMA transfer latency overhead could reach up to 325%, with a miss rate close to 85%. In contrast, after modifying the parameters of microarchitectural structures to enhance them (e.g., x8 DDTC entries, x32 IOTLB entries, and MSI translation), miss rates dropped to near 0%, resulting in negligible translation latency. Notably, IOTLB misses have a higher impact on added latency than DDTC misses, as each nested memory walk requires (at worst) 15 memory accesses to locate the final physical address. Nevertheless, the impact of DDTC misses increments when MSI translation support is enabled since the CDW requires more memory accesses to fetch the entire DC.

b) System-level Benchmarking

To conduct a more comprehensive evaluation of the IOMMU IP, we measured translation latencies for more sophisticated software setups: (i) Linux; and (ii) Bao running a single baremetal guest. Figure 6.32 shows the performance overhead introduced by the IOMMU in both configurations regarding the average transfer time of 1000 DMA transfers using five different microarchitectural configurations. These results corroborate the ones depicted in Figure 6.31 in that there is a similar pattern for both Linux and Bao setups, despite the differences in the absolute transfer time due to the Linux internals complexity.

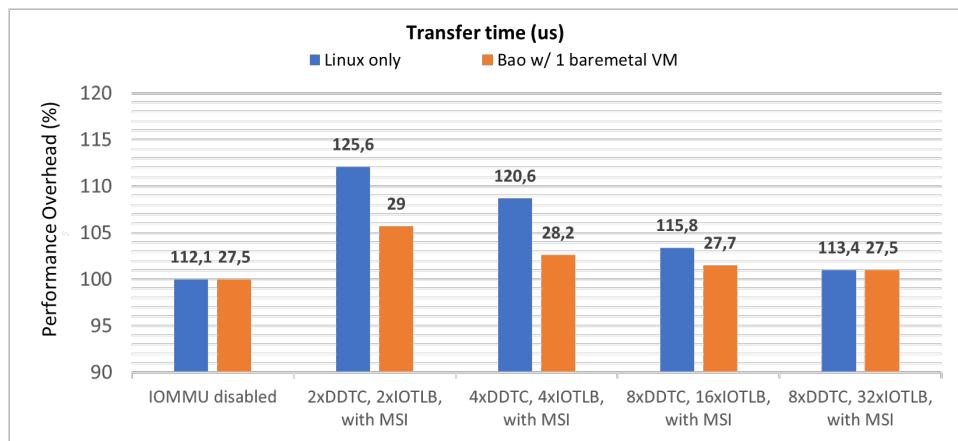


Figure 6.32: Average transfer time of 1000 DMA transfers in the system-level software setups using various IOMMU microarchitectural configurations. Given the substantial differences in absolute transfer times between both setups, the values were normalized.

Chapter 7

Conclusion

In this Master Thesis, we designed and implemented the first open-source IOMMU IP fully compliant with the recently ratified RISC-V IOMMU specification v1.0. The IP encompasses all mandatory features specified in the standard and incorporates some optional features as required by the target application. We defined a set of design parameters to configure IOMMU architectural features and microarchitectural properties at design time, achieving a scalable and adaptable IOMMU IP. Subsequently, we integrated the IP into a single-core CVA6-based SoC endowed with virtualization support. To validate and assess the IOMMU, we integrated four instances of the iDMA module and a DMA arbiter to mediate and route requests between the four iDMA devices and the translation request interface of the IOMMU. Furthermore, we extended the iDMA module with supplementary features required for our validation and evaluation setups. Concerning software, we added RISC-V IOMMU support in Bao hypervisor by implementing a software driver compliant with the architecture specification. This driver has already been upstreamed in the main repository of Bao [50].

Following the implementation phase, we conducted functional validation of the IOMMU IP, consisting of three gradual phases. Firstly, we developed a custom baremetal framework to perform granular testing of the supported architectural features. Secondly, we scaled up to a more realistic and comprehensive software setup based on Linux, using the RISC-V IOMMU driver developed by the RISC-V IOMMU task group. For this purpose, we developed a Linux device driver for the iDMA module and a user-space application to invoke the driver for issuing DMA transfers. Thirdly, we introduced a virtualization layer into the software stack, running Bao with a single Linux-based virtual machine. This phase aimed to test the RISC-V IOMMU driver developed for Bao and the IOMMU support for second-stage translation.

Upon successful completing the validation phase, we conducted an empirical evaluation concerning the operation and hardware resource utilization of the IOMMU IP within a virtualization-capable system. Leveraging the parameterizable nature of the IP, we expanded this analysis to cover various hardware con-

figurations based on different combinations of IOMMU architectural features (mandatory vs optional) and microarchitectural structures/parameters. We provided quantitative results regarding hardware costs and translation latencies, accompanied by the corresponding analysis. For example, using a baremetal setup with a naive microarchitecture configuration (e.g., 2 DDTC entries and 4 IOTLB entries), we observed a significant increase in DMA transfer latency ($\sim 325\%$), with a miss rate of approximately 85%. Nevertheless, after increasing key IOMMU microarchitectural structures (e.g., 8 DDTC entries and 32 IOTLB entries), miss rates dropped to near 0%, resulting in negligible translation latency. Moreover, we measured transfer times using system-level software setups (i.e., Linux and Bao hypervisor), and the results corroborated the values collected for the baremetal setup, i.e., a similar pattern for both Linux and Bao. All artifacts related to this work were open-sourced with permissive licenses in public repositories [49, 50, 52, 54, 55, 56], to foster collaboration within the RISC-V community.

7.1 Future Work

This work represents a pioneering effort in RISC-V and open-source hardware. We have successfully achieved a functional implementation compliant with the RISC-V IOMMU standard specification. Nonetheless, there is still room for improvement regarding the RISC-V IOMMU. For example, in Chapter 4, we identified the architectural features included in our design, chosen based on the intended application. However, some optional features were not included and remained as future work to complement the IOMMU IP. For instance, the IOMMU specification defines support for a register-based debug interface for explicit address translation of a user-defined IOVA (see Section 3.4.3). Furthermore, other features like MRIF support for MSI translation and those related to PCIe are candidates for future development.

At the software side, further efforts may be focused on improving the RISC-V IOMMU driver developed for Bao hypervisor. The current version of the driver implements support for second-stage translation and fault signaling. However, the architecture specification allows the hypervisor to provide a virtual IOMMU facility to guest OSs through hardware emulation or using a software interface. This feature would enable guest OSs to benefit from intra-guest protection using first-stage translation.

These suggestions are based on architectural features not implemented in this work, as they were considered non-critical for the target application. Based on the knowledge and experience acquired throughout the execution of this thesis, we discuss in the remainder of this chapter some alternative designs and architectural/microarchitectural changes to further optimize and enhance the developed IOMMU IP and the underlying RISC-V IOMMU specification.

Microarchitectural Improvements. To increase the IOMMU IP's throughput, we propose a pipelined design for address translations. We envision this implementation by splitting the CDW into two standalone modules (so that the DDT and PDT can be walked concurrently) and implementing three dedicated memory interfaces: one to fetch DCs, the other to fetch PCs, and the last one for the PTW. Additionally, we suggest the inclusion of store buffers within the IOMMU to reduce latency on write operations, along with an alternative mechanism to report write errors of buffered requests to devices. Moreover, to address potential IOMMU vulnerabilities to IOTLB-based side-channel attacks, like those documented in [57], we propose incorporating defense mechanisms, e.g., an internal attack detector using the IOMMU's HPM to identify suspicious DMA access patterns.

Architectural Enhancements. We believe there is a limitation in the RISC-V IOMMU specification, which prevents associating multiple DMA devices to the same DC. This feature would benefit hypervisors employing only second-stage translation and using the same page tables, VMID, and translation configuration for various devices (such as Bao), eliminating redundant DDT entries. As a possible solution, we suggest modifying the context mapping scheme with additional metadata, e.g., applying a bit mask in the *device_id* to map different devices to the same DC. In addition, we suggest rethinking some in-memory data structures for better memory utilization and granularity. For instance, segregating MSI translation data housed within the DC into a separate structure enables efficient sharing of the IOMMU between MSI-capable and MSI-incapable devices without wasting the 32 bytes reserved for MSI fields in each DC. Furthermore, separating first and second-stage translation data into different structures results in a smaller DC with only second-stage data, reducing latency during DDT walks and facilitating the implementation of a software interface for intra-guest protection.

Distributed IOMMU Architecture. The developed IOMMU IP implements a naive monolithic architecture, where the internal translation logic includes context/page table walkers and the IOATCs. This design option generates contention among devices as IOATCs are shared between them. Therefore, as future work, it is possible to implement a distributed IOMMU architecture, i.e., a dedicated IOATC per device (or group of devices) and a central translation unit containing the memory walkers. The main challenge is to conceive an architecture on the sweet spot between (i) reducing contention and translation latencies, (ii) increasing overall hardware costs by adding more IOATCs and other components, and (iii) handling the complexity introduced by using communication protocols between decentralized components and the central translation unit.

References

- [1] Sandro Pinto and Nuno Santos. “Demystifying Arm TrustZone: A Comprehensive Survey”. In: *ACM Computing Surveys* (2019).
- [2] Daniel Oliveira, Tiago Gomes, and Sandro Pinto. “uTango: An Open-Source TEE for IoT Devices”. In: *IEEE Access* (2022).
- [3] Bruno Sá, Francisco Marques, Manuel Rodriguez, José Martins, and Sandro Pinto. “Holistic RISC-V Virtualization: CVA6-Based SoC”. In: *Proceedings of the 20th ACM International Conference on Computing Frontiers*. 2023.
- [4] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. “Memory Access Control in Multiprocessor for Real-Time Systems with Mixed Criticality”. In: *24th Euromicro Conference on Real-Time Systems*. 2012.
- [5] Bruno Sá, Luca Valente, José Martins, Davide Rossi, Luca Benini, and Sandro Pinto. “CVA6 RISC-V Virtualization: Architecture, Microarchitecture, and Design Space Exploration”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2023).
- [6] Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, Leendert van Doorn, and Doorn Amd. “The price of safety: Evaluating IOMMU performance”. In: *Ottawa Linux Symposium (OLS)* (2007).
- [7] Intel Corporation. *Intel® Virtualization Technology for Directed I/O Architecture Specification. Revision 4.1*. 2023. URL: <https://www.intel.com/content/www/us/en/content-details/774206/intel-virtualization-technology-for-directed-i-o-architecture-specification.html>.
- [8] Advanced Micro Devices Inc. *AMD I/O Virtualization Technology (IOMMU) Specification. Revision 3.07-PUB*. 2022. URL: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/specifications/48882_IOMMU.pdf.

- [9] Arm Limited Company. *Arm System Memory Management Unit Architecture Specification. Version D.b.* 2021. URL: <https://developer.arm.com/documentation/ih10070/db/?lang=en>.
- [10] Krste Asanović and David A. Patterson. *Instruction Sets Should Be Free: The Case For RISC-V.* Tech. rep. EECS Department, University of California, Berkeley, 2014. URL: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html>.
- [11] RISC-V IOMMU Task Group. *RISC-V IOMMU Specification Document. Version 1.0.* 2023. URL: <https://github.com/riscv-non-isa/riscv-iommu/blob/v1.0/riscv-iommu.pdf>.
- [12] Cristiano Rodrigues, Ivo Marques, Sandro Pinto, Tiago Gomes, and Adriano Tavares. "Towards a Heterogeneous Fault-Tolerance Architecture based on Arm and RISC-V Processors". In: *IECON - 45th Annual Conference of the IEEE Industrial Electronics Society*. 2019.
- [13] José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto. "Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems". In: *Workshop on Next Generation Real-Time Embedded Systems (NG-RES)*. 2020.
- [14] David Harris and Sarah Harris. *Digital Design and Computer Architecture, Second Edition*. 2nd. Morgan Kaufmann Publishers Inc., 2012. ISBN: 0123944244.
- [15] David A. Patterson and John L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. 1st. Morgan Kaufmann Publishers Inc., 2017. ISBN: 0128122757.
- [16] Michael J. Flynn and Kevin W. Rudd. "Parallel Architectures". In: *ACM Computing Surveys* (1996).
- [17] Andrew Waterman, Krste Asanović, and John. Hauser. *The RISC-V Instruction Set Manual. Volume II: Privileged Architecture v20211203*. 2021. URL: https://drive.google.com/file/d/1EMip5dZlnypTk7pt4WWUKmtjUKT0kBqh/view?usp=drive_link.
- [18] Florian Zaruba and Luca Benini. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2019).
- [19] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual. Volume I: Unprivileged ISA v20191213*. 2019. URL: https://drive.google.com/file/d/1s01ZxUZaa7eV_00_WsZzaurFLLww7ou5/view?usp=drive_link.

- [20] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. *The Rocket Chip Generator*. Tech. rep. EECS Department, University of California, Berkeley, 2016. URL: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html>.
- [21] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. 4th. Prentice Hall Press, 2014. ISBN: 013359162X.
- [22] José Martins and Sandro Pinto. “Shedding Light on Static Partitioning Hypervisors for Arm-based Mixed-Criticality Systems”. In: *IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2023.
- [23] Gernot Heiser. “The Role of Virtualization in Embedded Systems”. In: *1st Workshop on Isolation and Integration in Embedded Systems*. 2008.
- [24] Gerald J. Popek and Robert P. Goldberg. “Formal Requirements for Virtualizable Third Generation Architectures”. In: *Communications of the ACM* (1974).
- [25] Tiago Gomes, Filipe Salgado, Jorge Cabral, Adriano Tavares, and João Monteiro. “Non-Intrusive Hardware Acceleration for Dynamic Binary Translation in Embedded Systems”. In: *IEEE International Conference on Industrial Technology (ICIT)*. 2019.
- [26] Robert P. Goldberg. “Architectural Principles for Virtual Computer Systems”. PhD thesis. Harvard University, 1972.
- [27] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. “Xen and the Art of Virtualization”. In: *19th ACM Symposium on Operating Systems Principles*. 2003.
- [28] Anup Patel, Mai Daftedar, Mohamed Shalan, and M. Watheq El-Kharashi. “Embedded Hypervisor Xvisor: A Comparative Analysis”. In: *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 2015.
- [29] Jordan Shropshire. “Analysis of Monolithic and Microkernel Architectures: Towards Secure Hypervisor Design”. In: *47th Hawaii International Conference on System Sciences*. 2014.

- [30] Bruno Sá, José Martins, and Sandro Pinto. "A First Look at RISC-V Virtualization From an Embedded Systems Perspective". In: *IEEE Transactions on Computers* (2022).
- [31] Abhishek Bhattacharjee and Daniel Lustig. *Architectural and Operating System Support for Virtual Memory*. Springer International Publishing, 2017. ISBN: 9783031006296.
- [32] Éric Lacombe, Vincent Nicomette, and Yves Deswarte. "Enforcing Kernel Constraints by Hardware-Assisted Virtualization". In: *Journal of Computer Virology* (2011).
- [33] Fernand Lone Sang, Éric Lacombe, Vincent Nicomette, and Yves Deswarte. "Exploiting an I/OMMU vulnerability". In: *5th International Conference on Malicious and Unwanted Software*. 2010.
- [34] Keith Adams and Ole Agesen. "A Comparison of Software and Hardware Techniques for X86 Virtualization". In: *SIGOPS Oper. Syst. Rev.* (2006).
- [35] A. Golchin, S. Sinha, and R. West. "Boomerang: Real-Time I/O Meets Legacy Systems". In: *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2020.
- [36] Eric Missimer, Katherine Missimer, and Richard West. "Mixed-Criticality Scheduling with I/O". In: *28th Euromicro Conference on Real-Time Systems (ECRTS)*. 2016.
- [37] Matthew Danish, Ye Li, and Richard West. "Virtual-CPU Scheduling in the Quest Operating System". In: *17th IEEE Real-Time and Embedded Technology and Applications Symposium*. 2011.
- [38] Alejandro Masrur, Sebastian Drossler, Thomas Pfeuffer, and Samarjit Chakraborty. "VM-Based Real-Time Services for Automotive Control Applications". In: *IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*. 2010.
- [39] Oliver Schwarz and Christian Gehrman. "Securing DMA through virtualization". In: *Complexity in Engineering (COMPENG). Proceedings*. 2012.
- [40] Emiliano Betti, Stanley Bak, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. "Real-Time I/O Management System with COTS Peripherals". In: *IEEE Transactions on Computers* (2013).
- [41] Daniel Muench, Michael Paulitsch, and Andreas Herkersdorf. "IOMPU: Spatial Separation for Hardware-Based I/O Virtualization for Mixed-Criticality Embedded Real-Time Systems Using Non-transparent Bridges". In: *IEEE 17th International Conference on High Performance Computing and Communications*. 2015.
- [42] Zhe Jiang, Neil Audsley, Pan Dong, Nan Guan, Xiaotian Dai, and Lifeng Wei. "MCS-IOV: Real-Time I/O Virtualization for Mixed-Criticality Systems". In: *IEEE Real-Time Systems Symposium (RTSS)*. 2019.

- [43] Govindarajan Sriramakrishnan, Mihir Mody, Gregory Shurtz, Chuck Fuoco, Kedar Chitnis, Nikhil Devshatwar, Don Steiss, Jonathan Bergsagel, Jason Jones, and Prithvi Shankar Y A. "IO Virtualization for Real Time Automotive Systems". In: *IEEE International Conference on Consumer Electronics (ICCE)*. 2022.
- [44] Sandro Pinto, Jorge Pereira, Tiago Gomes, Adriano Tavares, and Jorge Cabral. "LTZVisor: TrustZone is the Key". In: *Euromicro Conference on Real-Time Systems*. 2017.
- [45] Zhe Jiang and Neil Audsley. "BlueIO: A Scalable Real-Time Hardware I/O Virtualization System for Many-core Embedded Systems". In: *ACM Transactions in Embedded Computing Systems* (2019).
- [46] John Hauser. *The RISC-V Advanced Interrupt Architecture, Version 1.0*. 2023. URL: <https://github.com/riscv/riscv-ai/releases/download/1.0/riscv-interrupts-1.0.pdf>.
- [47] PCI SIG. *PCIe Base Specification Revision 6.1*. 2023. URL: <https://pcisig.com/pci-express-6.0-specification>.
- [48] Arm Limited or its affiliates. *AMBA AXI Protocol Specification, Issue J*. 2023. URL: <https://developer.arm.com/documentation/ih10022/j/>.
- [49] *GitHub repository of the RISC-V IOMMU IP*. URL: <https://github.com/zero-day-labs/riscv-iommu>.
- [50] *GitHub repository of Bao hypervisor*. URL: <https://github.com/bao-project/bao-hypervisor>.
- [51] Luca Valente, Asif Veeran, Mattia Sinigaglia, Yvan Tortorella, Alessandro Nadalini, Nils Wistoff, Bruno Sá, Angelo Garofalo, Rafail Psiakis, M. Tolba, Ari Kulmala, Nimisha Limaye, Ozgur Sinanoglu, Sandro Pinto, Daniele Palossi, Luca Benini, Baker Mohammad, and Davide Rossi. "Shaheen: An Open, Secure, and Scalable RV64 SoC for Autonomous Nano-UAVs". In: *IEEE Hot Chips 35 Symposium (HCS)*. 2023.
- [52] *GitHub repository of the CVA6-based SoC w/ IOMMU*. URL: <https://github.com/zero-day-labs/cva6/tree/feat/iommu>.
- [53] Thomas Benz, Michael Rogenmoser, Paul Scheffler, Samuel Riedel, Alessandro Ottaviano, Andreas Kurth, Torsten Hoefler, and Luca Benini. "A High-performance, Energy-efficient Modular DMA Engine Architecture". In: *Computing Research Repository (CoRR)* (2023).

- [54] *GitHub repository of the iDMA module with modifications*. URL: <https://github.com/malejo97/iDMA>.
- [55] *GitHub repository of the IOMMU baremetal test framework*. URL: <https://github.com/zero-day-labs/riscv-iommu-tests>.
- [56] *GitHub repository of the Linux-based software setup*. URL: <https://github.com/malejo97/linux/tree/tjeznach/riscv-iommu-aia>.
- [57] Taehun Kim, Hyeongjin Park, Seokmin Lee, Seunghee Shin, Junbeom Hur, and Youngjoo Shin. “DEVIOUS: Device-Driven Side-Channel Attacks on the IOMMU”. In: *IEEE Symposium on Security and Privacy (SP)*. 2023.