

POLITECNICO DI TORINO

Master's Degree in Electronics Engineering



**Politecnico
di Torino**

Master's Degree Thesis

Design and implementation of a RISC-V processor including security features

Supervisors

Prof. Edgar Ernesto SANCHEZ

Prof. Stefano DI CARLO

Candidate

Behnam FARNAGHINEJAD

October 2024

Abstract

In an age where digital security is paramount, the development of secure and efficient processors is crucial for safeguarding sensitive information and ensuring the integrity of computing systems. As cyber threats evolve in sophistication, there is an increasing demand for hardware-level security features that can provide robust defenses against various attacks.

One effective way to make processors more secure is to include special instructions directly in the hardware. This thesis focuses on designing and making a RISC-V processor that has these security features. The goal is to significantly improve the processor's capability to execute cryptography tasks efficiently and securely, leveraging the newly ratified RISC-V Cryptography Extensions. This research holds substantial significance as it advances secure processor design, which is critical for applications ranging from personal computing to large-scale data centers and vital infrastructure protection.

The research begins by selecting SystemVerilog as the preferred hardware description language and evaluating various 64-bit cores for compatibility with Linux. Ultimately, the CVA6 is chosen as the optimal platform. This CPU adheres to the 64-bit RISC-V instruction set and supports multiple extensions and three levels of user access similar to Unix systems. The study also thoroughly investigates the RISC-V Cryptography Extensions Volume I to establish the project's foundation. It involves meticulously designing a cryptography accelerator within the processor, followed by rigorous testing phases that include functional testing, spike simulation validation, and comprehensive regression testing to ensure reliability. Extensive code coverage analysis validates the effectiveness of the test suite.

The culmination of these efforts results in the successful integration of the cryptography accelerator as a co-processor within the CVA6 core, significantly enhancing its security capabilities and extending its functionality. For instance, the AES encryption algorithm demonstrates performance improvements with speed gains of approximately 94% and reduced code size. In decryption, a 98% reduction in execution time is observed, along with a decrease in code size, significantly enhancing the implementation security of cryptography algorithms through hardware-based computation.

The thesis concludes with a thorough analysis of the results, underscoring the contributions' significance and suggesting avenues for future research and development. Hardware implementations inherently enhance security by isolating critical functions from the main processor, thereby reducing vulnerability to malicious

software and mitigating certain types of attacks like side-channel threats. Moreover, hardware implementations offer constant-time execution for cryptography algorithms, further bolstering security measures.

Acknowledgements

I want to thank my thesis supervisors, Prof. Edgar Ernesto Sanchez and Prof. Stefano Di Carlo, for their great help, guidance, and useful feedback during my research. Their knowledge and encouragement were really important for this thesis.

I am very grateful to my family for their constant love and support throughout this academic journey. Their support means everything to me.

“Behnam”

Table of Contents

List of Tables	VI
List of Figures	VII
Acronyms	IX
1 Introduction	1
1.1 Background	1
1.2 Motivation	2
1.3 Objectives	2
2 Literature Review	3
2.1 RISC-V Architecture Overview	3
2.2 Security Features in RISC-V	3
2.2.1 Privileged Modes: User and Supervisor Modes	4
2.2.2 Physical Memory Protection (PMP)	4
2.2.3 Cryptography Extensions	4
2.3 CVA6 Core	5
3 Methodology	7
3.1 Selection of HDL Language	7
3.2 Evaluation of 64-bit Cores	7
3.3 Exploration of Cryptography Extensions	8
3.4 Development of Cryptography Accelerator	10
3.5 Integration with CVA6 Core	12
3.6 Testing and Validation	13
4 Design and Implementation	14
4.1 CVA6 Core Overview	14
4.1.1 Pipeline Stages	14
4.1.2 PMP	17

4.1.3	PMA	18
4.2	Core-V eXtension InterFace (CV-X-IF)	19
4.2.1	Features	20
4.2.2	Parameters	20
4.2.3	Interfaces	20
4.2.4	Operating Principle	21
4.3	Cryptography Accelerator	24
4.3.1	AES Encryption/Decryption	28
4.3.2	SHA-2 Hash Functions	33
4.3.3	SM4 and SM3 Functions	37
5	Testing and Validation	40
5.1	Functional Testing of Modules	40
5.2	Regression Test and Code Coverage Analysis	41
5.3	Summary	43
6	Results and Discussion	44
6.1	Area Report	44
6.2	Power Report	45
6.3	Performance Analysis	48
6.4	Discussion	49
6.4.1	Design Trade-offs	49
6.4.2	Security Implications	50
7	Conclusion and Future Work	51
7.1	Conclusion	51
7.2	Future Work	52
	Bibliography	54

List of Tables

2.1	Supported combination of privilege modes	4
2.2	List of Scalar Cryptography Extensions	5
3.1	NIST Suite Assembly Instructions	9
3.2	ShangMi Suite Assembly Instructions	9
3.3	Other useful assembly instructions for Cryptography	10
3.4	Detailed Decoding of Cryptography Instructions	11
4.1	Supported CVXIF Parameters in CVA6	21
5.1	Simulation Results of Regression Tests	42
6.1	Comparison of Area Metrics between CVA6 and Cryptography Accelerator	45
6.2	CVA6 Power Report (With and Without Accelerator)	47
6.3	Cryptography Accelerator Power Report (Encryption and Decryption)	47
6.4	Performance Analysis of AES with/without Cryptography Extensions	49

List of Figures

4.1	CVA6 core overview	15
4.2	Interfaces in CV-X-IF	22
4.3	Operating Principle of CVXIF	22
4.4	CVA6 + Cryptography Accelerator	26
4.5	Cryptography Accelerator Block Diagram	27
4.6	Key Expansion Phase of AES Encryption	29
4.7	AES Encryption Phase for One Block	30
5.1	Waveforms of the SHA2 family results	40
5.2	Code Coverage Results	41

Acronyms

AES

Advanced Encryption Standard

ALU

Arithmetic Logic Unit

ASIC

Application-Specific Integrated Circuit

RISC

Complex Instruction Set Computer

CPU

Central Processing Unit

CSR

Control and Status Register

CVXIF

Core-V eXtension InterFace

D\$

Data Cache

FPGA

Field-Programmable Gate Array

FPU

Floating Point Unit

FU
Functional Unit

hart
Hardware Thread

HDL
Hardware Description Language

I\$
Instruction Cache

IP
Intellectual Property Block

ISA
Instruction Set Architecture

L1
Level 1 Cache

LSU
Load Store Unit

MMU
Memory Management Unit

PC
Program Counter

PMP
Physical Memory Protection

PMP
Physical Memory Attribute

PTW
Page Table Walk

RISC

Reduced Instruction Set Computer

RTL

Register Transfer Level

SHA

Secure Hash Algorithm

SPIKE

RISC-V ISA Simulator

TLB

Translation Lookaside Buffer

WT

Write-Through

WT

Write-Enable

Chapter 1

Introduction

1.1 Background

Processor design is fundamental to the advancement of computing technology. At its core, a processor executes instructions provided by software, translating them into actions that drive hardware operations. Instruction Set Architectures (ISAs) define the set of instructions that a processor can execute, playing a crucial role in determining a processor's functionality and performance. Among the various ISAs, the Reduced Instruction Set Computer (RISC) architecture has been influential, emphasizing simplicity and efficiency.

The evolution of processor architectures has seen significant milestones, from early complex instruction set computers (CISC) to the development of RISC. RISC-V, an open-source ISA, has emerged as a transformative force in this landscape. Unlike proprietary ISAs, RISC-V is designed to be extensible and customizable, allowing for innovation and adaptation to diverse applications.

The importance of processor security has never been more pronounced, as evidenced by vulnerabilities such as side-channel attacks, which have exposed significant flaws in widely used architectures. As the computing landscape continues to evolve, the need for robust, secure processors becomes paramount. These incidents highlight the necessity of integrating security features at the hardware level to protect against various attack vectors.

This thesis explores the design and implementation of a 64-bit RISC-V processor, integrating advanced security features to address contemporary cybersecurity threats.

1.2 Motivation

Proprietary ISAs often limit innovation and customization due to restricted access and licensing constraints. In contrast, open-source ISAs like RISC-V provide a flexible foundation for developing tailored solutions. The ability to modify and extend the ISA facilitates advancements in processor design and encourages community-driven improvements.

Recent cybersecurity incidents have revealed significant vulnerabilities in existing processor designs, demonstrating the urgent need for secure processors. Current designs often lack comprehensive security features, leaving systems susceptible to attacks. By incorporating advanced security mechanisms, processors can be made more resilient against emerging threats.

The motivation for this research stems from the growing need for secure and efficient processing systems. Traditional software-based security measures, while effective, can be vulnerable to various forms of attacks, such as side-channel attacks and timing attacks. By incorporating cryptography features directly into the hardware, processors can achieve a higher level of security and efficiency.

1.3 Objectives

The primary objective of this thesis is to design and implement a RISC-V processor with integrated security features. Specifically, this research focuses on incorporating the newly ratified RISC-V Cryptography Extensions into a 64-bit core. The goals of this project include:

- Selecting a suitable available open-source RISC-V core to enhance it for the goal, rather than designing from scratch.
- Implementing scalar cryptography extensions and integrating them into the selected core.
- Conducting rigorous testing and validation to ensure the reliability and effectiveness of the implemented hardware.
- Analyzing the performance improvements and security enhancements achieved through the hardware-based cryptography extensions.

This research contributes to the field of secure processor design and aims to provide a foundation for future developments in hardware-based security mechanisms. The successful implementation of these features will not only enhance the security capabilities of the core but also demonstrate the potential of hardware-level security measures in protecting sensitive information across various applications.

Chapter 2

Literature Review

2.1 RISC-V Architecture Overview

The RISC-V architecture is an open-source hardware instruction set architecture (ISA) that offers extensibility and simplicity. Designed to support a wide range of devices, from small embedded systems to high-performance computing platforms, RISC-V's open nature allows for widespread academic and industry collaboration, fostering innovation and rapid development.

RISC-V's modular design includes a base integer instruction set (I), which can be extended with optional extensions to provide additional functionality. These extensions include, but are not limited to, multiplication and division operations (M), atomic instructions (A), floating-point operations (F and D), and various other specialized capabilities. The extensibility of RISC-V is one of its key strengths, allowing designers to tailor processors to specific application needs without unnecessary complexity.

The architecture supports both 32-bit (RV32) and 64-bit (RV64) address spaces, as well as a 128-bit (RV128) version for future scalability. This flexibility enables RISC-V to be employed in a diverse array of computing environments, from low-power embedded devices to powerful server-grade processors.[1]

2.2 Security Features in RISC-V

Security in modern processors is critical due to the increasing prevalence of cyber threats. RISC-V addresses this need by incorporating various security features directly into the hardware. This section will discuss privileged modes, Physical Memory Protection (PMP) and cryptography extensions as key components of RISC-V's security architecture.[2]

2.2.1 Privileged Modes: User and Supervisor Modes

In RISC-V, each hardware thread (hart) operates at a specific privilege level, managed by Control and Status Registers (CSRs). RISC-V defines three primary privilege levels: Machine (M), Supervisor (S), and User/Application (U). Table 2.1 is shown different levels and modes:

Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems

Table 2.1: Supported combination of privilege modes

These levels ensure protection between software components. Code usually runs in U-mode until a trap (e.g., a supervisor call) forces a switch to a higher privilege mode (e.g., S-mode). M-mode, with the highest privileges, is the only mandatory level used for trusted code with full access to the machine. Systems may implement one to three modes, balancing isolation and complexity.

M-mode, always implemented, grants full access to the machine. Simple systems may only use M-mode, while more complex implementations add U-mode and S-mode for better isolation and security.[2]

2.2.2 Physical Memory Protection (PMP)

To ensure secure processing and fault containment, RISC-V incorporates an optional Physical Memory Protection (PMP) unit. PMP allows the definition of access privileges (read, write, execute) for specific physical memory regions on a per-hart basis. These privileges are enforced through machine-mode control registers. PMP checks are conducted in parallel with Physical Memory Attribute (PMA) checks, and apply to memory accesses in Supervisor (S) and User (U) modes, as well as certain Machine (M) mode configurations. PMP ensures that unauthorized accesses are trapped precisely, preventing potential security breaches.[2]

2.2.3 Cryptography Extensions

The RISC-V Cryptography Extensions are a significant step towards enhancing security at the processor level. The Scalar Cryptography Extensions (Volume I) introduce a set of instructions specifically designed for cryptography algorithms, such as encryption, decryption and hash functions. These instructions enable the processor to perform cryptography operations more efficiently than traditional

software implementations. The proposed standard Scalar Cryptography extensions to the RISC-V ISA are: [3]

Instruction	Description
Zknd	NIST Suite: AES Decryption
Zkne	NIST Suite: AES Encryption
Zknh	NIST Suite: Hash Function Instructions
Zksed	ShangMi Suite: SM4 Block Cipher Instructions
Zksh	ShangMi Suite: SM3 Hash Function Instructions
Zkr	Entropy Source Extension
Zkn	NIST Algorithm Suite (Zknd, Zkne, Zknh, Zbkb, Zbkc, Zbkx)
Zks	ShangMi Algorithm Suite (Zksed, Zksh, Zbkb, Zbkc, Zbkx)
Zk	Standard scalar cryptography extension (Zkn, Zkt, Zkr)
Zkt	Data Independent Execution Latency
Zbkb	Bitmanip instructions for Cryptography
Zbkc	Carry-less multiply instructions
Zbkx	Crossbar permutation instructions

Table 2.2: List of Scalar Cryptography Extensions

These hardware-accelerated instructions provide several benefits, including reduced execution time, lower power consumption, and enhanced resistance to certain types of attacks, such as timing and side-channel attacks.

2.3 CVA6 Core

The CVA6 core, formerly known as Ariane, is a 64-bit application-class RISC-V CPU designed for high performance and scalability. It adheres to the RV64GC ISA, which includes the general-purpose extensions (I, M, A, F, D, and C) and additional custom extensions as needed. The CVA6 core is suitable for running operating systems like Linux, making it a versatile choice for various applications. Key features of the CVA6 core include:[4]

- Implements three privilege levels M, S, U to fully support a Unix-like operating system
- 6-stage pipeline, 32- or 64-bit, In-order issue, Out-Of-Order Execution, In-Order Commit
- L1 instruction and data caches, as well as an optional L2 caches
- Configurable size, separate TLBs, a hardware PTW and branch prediction (BTB and BHT)

- Optional features such as MMU, PMP, FPU, cache organization and size and etc.
- Implements the CVXIF interface to tightly integrate co-processors into the execution stage

The combination of these features makes the CVA6 core a robust platform for implementing and evaluating the RISC-V cryptography extensions as a co-processor.

Chapter 3

Methodology

3.1 Selection of HDL Language

The choice of hardware description language (HDL) is a critical step in designing and implementing the RISC-V processor with security features. For this project, SystemVerilog was chosen due to its extensive feature set, which extends Verilog with advanced system-level design capabilities. SystemVerilog offers enhanced data types, concurrency mechanisms, and powerful verification tools, making it particularly suitable for complex hardware designs.

Another key factor in selecting SystemVerilog is that the chosen core, the CVA6 core, is written in this language, ensuring compatibility and facilitating integration with the existing design.

3.2 Evaluation of 64-bit Cores

To select the most suitable 64-bit core for integrating cryptography extensions, several RISC-V cores were evaluated based on compatibility, performance, and Linux support. The evaluation criteria included:

- **ISA Compliance:** Adherence to the RV64GC ISA, ensuring support for general-purpose instructions.
- **Performance:** Assessment of core performance in terms of processing speed, efficiency, and support for out-of-order execution.
- **Scalability:** Capability to support advanced features like virtual memory and multi-core configurations.
- **Community and Documentation:** Availability of comprehensive documentation and an active development community for ongoing support and improvements.

After a thorough evaluation, the CVA6 core was selected due to its high performance, extensibility, and active community support. Its compliance with the RV64GC ISA and robust feature set make it an ideal platform for this project. Additionally, the CVA6 core includes the CVXIF interface, which allows for the tight integration of a co-processor at the execution stage without requiring modifications to the main core's RTL design. In this project, the cryptography extension will be implemented by designing the accelerator as a co-processor using this interface.

3.3 Exploration of Cryptography Extensions

The next step involved a detailed study of the RISC-V Cryptography Extensions Volume I. These extensions define a set of instructions specifically designed for cryptography operations, including:

- NIST Suite: Instructions for AES encryption and decryption, SHA2-256, and SHA2-512 (Table 3.1)
- ShangMi Suite: Instructions for the SM4 block cipher and SM3 hash function (Table 3.2)
- Other useful cryptography instructions: Including Bitmanip, Carry-less multiply, and Crossbar permutation (Table 3.3)

Understanding these extensions was essential for designing the cryptography accelerator. The study focused on the implementation details, expected performance benefits, and potential security improvements provided by these hardware-accelerated instructions.

The following tables summarize the assembly instructions included in these cryptography extensions:

RV32	RV64	Assembly Instruction	Extension
*		aes32dsi rd, rs1, rs2, bs	Zknd
*		aes32dsmi rd, rs1, rs2, bs	Zknd
	*	aes64ds rd, rs1, rs2	Zknd
	*	aes64dsm rd, rs1, rs2	Zknd
	*	aes64im rd, rs1	Zknd
	*	aes64ks2 rd, rs1, rs2	Zknd/Zkne
	*	aes64ks1i rd, rs1, rnum	Zknd/Zkne
*		aes32esi rd, rs1, rs2, bs	Zkne
*		aes32esmi rd, rs1, rs2, bs	Zkne
	*	aes64es rd, rs1, rs2	Zkne
	*	aes64esm rd, rs1, rs2	Zkne
*	*	sha256sig0 rd, rs1	Zknh
*	*	sha256sig1 rd, rs1	Zknh
*	*	sha256sum0 rd, rs1	Zknh
*	*	sha256sum1 rd, rs1	Zknh
*		sha512sig0h rd, rs1, rs2	Zknh
*		sha512sig0l rd, rs1, rs2	Zknh
*		sha512sig1h rd, rs1, rs2	Zknh
*		sha512sig1l rd, rs1, rs2	Zknh
*		sha512sum0r rd, rs1, rs2	Zknh
*		sha512sum1r rd, rs1, rs2	Zknh
	*	sha512sig0 rd, rs1	Zknh
	*	sha512sig1 rd, rs1	Zknh
	*	sha512sum0 rd, rs1	Zknh
	*	sha512sum1 rd, rs1	Zknh

Table 3.1: NIST Suite Assembly Instructions

RV32	RV64	Assembly Instruction	Extension
*	*	sm4ed rd, rs1, rs2, bs	Zksed
*	*	sm4ks rd, rs1, rs2, bs	Zksed
*	*	sm3p0 rd, rs1	Zksh
*	*	sm3p1 rd, rs1	Zksh

Table 3.2: ShangMi Suite Assembly Instructions

RV32	RV64	Assembly Instruction	Extension
*	*	andn rd, rs1, rs2	Zbkb
*	*	brev8 rd, rs	Zbkb
*	*	orn rd, rs1, rs2	Zbkb
*	*	pack rd, rs1, rs2	Zbkb
*	*	packh rd, rs1, rs2	Zbkb
	*	packw rd, rs1, rs2	Zbkb
*	*	rev8 rd, rs	Zbkb
*	*	rol rd, rs1, rs2	Zbkb
	*	rolw rd, rs1, rs2	Zbkb
*	*	ror rd, rs1, rs2	Zbkb
*	*	rori rd, rs1, shamt	Zbkb
	*	roriw rd, rs1, shamt	Zbkb
	*	rorw rd, rs1, rs2	Zbkb
*	*	xnor rd, rs1, rs2	Zbkb
*		zip rd, rs	Zbkb
*		unzip rd, rs	Zbkb
*	*	clmul rd, rs1, rs2	Zbkc
*	*	clmulh rd, rs1, rs2	Zbkc
*	*	xperm8 rd, rs1, rs2	Zbkx
*	*	xperm4 rd, rs1, rs2	Zbkx

Table 3.3: Other useful assembly instructions for Cryptography

3.4 Development of Cryptography Accelerator

The development of the cryptography extensions for the RISC-V architecture was carried out in multiple stages, each focusing on different aspects of the implementation process.

The key steps in this development process included:

- **Instruction Decoding:** The cryptography instructions were decoded based on the specifications illustrated in Table 3.4.

Assembly Instruction	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
aes32dsi	rd, rs1, rs2, bs	bs	1	0	1	0	1	0	1	rs2					rs1		0	0	0	0	rd				0	1	1	0	0	1	1	
aes32dsmi	rd, rs1, rs2, bs	bs	1	0	1	1	1	1	1	rs2					rs1		0	0	0	0	rd				0	1	1	0	0	1	1	
aes64ds	rd, rs1, rs2	0	1	1	1	1	1	1	1	rs2					rs1		0	0	0	0	rd				0	1	1	0	0	1	1	
aes64dsm	rd, rs1, rs2	0	1	1	1	1	1	1	1	rs2					rs1		0	0	0	0	rd				0	1	1	0	0	1	1	
aes64im	rd, rs1	0	0	1	1	0	0	0	0	0	0	0	0	0	rs1		0	0	0	1	rd				0	1	1	0	0	0	1	
aes64ks2	rd, rs1, rs2	0	1	1	1	1	1	1	1	rs2					rs1		0	0	0	0	rd				0	1	1	0	0	0	1	
aes64ks1l	rd, rs1, rnum	0	1	1	1	0	0	0	1	rnum					rs1		0	0	0	1	rd				0	0	1	0	0	0	1	
aes32esi	rd, rs1, rs2, bs	bs	1	0	0	0	1	1	1	rs2					rs1		0	0	0	0	rd				0	1	1	0	0	0	1	
aes32esmi	rd, rs1, rs2, bs	bs	1	0	0	1	1	1	1	rs2					rs1		0	0	0	0	rd				0	1	1	0	0	0	1	
aes64es	rd, rs1, rs2	0	1	1	0	0	1	1	1	rs2					rs1		0	0	0	0	rd				0	1	1	0	0	0	1	
aes64esm	rd, rs1, rs2	0	1	1	0	0	1	1	1	rs2					rs1		0	0	0	0	rd				0	1	1	0	0	0	1	
sha256sig0	rd, rs1	0	0	0	1	0	0	0	0	0	0	0	0	0	rs1		0	0	1	0	rd				0	1	0	0	0	0	1	
sha256sig1	rd, rs1	0	0	0	1	0	0	0	0	0	0	1	1	1	rs1		0	0	1	0	rd				0	0	1	0	0	0	1	
sha256sum0	rd, rs1	0	0	0	1	0	0	0	0	0	0	0	0	0	rs1		0	0	0	0	rd				0	0	1	0	0	0	0	
sha256sum1	rd, rs1	0	0	0	1	0	0	0	0	0	0	0	0	0	rs1		0	0	0	0	rd				0	0	1	0	0	0	0	
sha512sig0h	rd, rs1, rs2	0	1	0	1	1	1	1	1	rs2					rs1		0	0	0	0	rd				0	1	1	0	0	0	0	
sha512sig0l	rd, rs1, rs2	0	1	0	1	0	1	1	1	rs2					rs1		0	0	0	0	rd				0	1	1	0	0	0	0	
sha512sig1h	rd, rs1, rs2	0	1	0	1	1	1	1	1	rs2					rs1		0	0	0	0	rd				0	1	1	0	0	0	0	
sha512sig1l	rd, rs1, rs2	0	1	0	1	0	1	1	1	rs2					rs1		0	0	0	0	rd				0	1	1	0	0	0	0	
sha512sum0r	rd, rs1, rs2	0	1	0	1	0	0	0	0	rs2					rs1		0	0	0	0	rd				0	1	1	0	0	0	0	
sha512sum1r	rd, rs1, rs2	0	1	0	1	0	0	0	0	rs2					rs1		0	0	0	0	rd				0	1	1	0	0	0	0	
sha512sig0	rd, rs1	0	0	0	1	0	0	0	0	0	0	0	0	0	rs1		0	0	0	1	rd				0	0	1	0	0	0	0	
sha512sig1	rd, rs1	0	0	0	1	0	0	0	0	0	0	1	1	1	rs1		0	0	0	1	rd				0	0	1	0	0	0	0	
sha512sum0	rd, rs1	0	0	0	1	0	0	0	0	0	0	0	0	0	rs1		0	0	0	0	rd				0	0	1	0	0	0	0	
sha512sum1	rd, rs1	0	0	0	1	0	0	0	0	0	0	0	0	0	rs1		0	0	0	0	rd				0	0	1	0	0	0	0	
sm4ed	rd, rs1, rs2, bs	bs	1	1	0	0	0	0	0	rs2					rs1		0	0	0	0	rd				0	1	1	0	0	0	0	
sm4ks	rd, rs1, rs2, bs	bs	1	1	0	0	0	0	0	rs2					rs1		0	0	0	0	rd				0	1	1	0	0	0	0	
sm3p0	rd, rs1	0	0	0	1	0	0	0	0	0	0	0	0	0	rs1		0	0	0	0	rd				0	0	1	0	0	0	0	
sm3p1	rd, rs1	0	1	0	0	0	0	0	0	0	0	0	0	0	rs1		0	0	0	0	rd				0	0	1	0	0	0	0	
andn	rd, rs1, rs2	0	1	0	0	0	0	0	0	rs2					rs1		1	1	1	1	rd				0	1	1	0	0	0	0	
brev8	rd, rs	0	1	1	0	0	0	0	0	0	0	0	0	0	rs1		1	1	1	1	rd				0	1	1	0	0	0	0	
dm	rd, rs1, rs2	0	1	0	0	0	0	0	0	rs2					rs1		1	1	1	1	rd				0	1	1	0	0	0	0	
pack	rd, rs1, rs2	0	0	0	0	1	0	0	0	rs2					rs1		1	1	1	1	rd				0	1	1	0	0	0	0	
packh	rd, rs1, rs2	0	0	0	0	1	0	0	0	rs2					rs1		1	1	1	1	rd				0	1	1	0	0	0	0	
packw	rd, rs1, rs2	0	0	0	0	1	0	0	0	rs2					rs1		1	1	1	1	rd				0	1	1	0	0	0	0	
rev8	rd, rs	0	1	1	0	1	0	0	0	rs2					rs1		1	1	1	1	rd				0	1	1	0	0	0	0	
rol	rd, rs1, rs2	0	1	1	0	0	0	0	0	rs2					rs1		1	1	1	1	rd				0	0	1	0	0	0	0	
rolw	rd, rs1, rs2	0	1	1	0	0	0	0	0	rs2					rs1		0	0	0	0	rd				0	0	1	0	0	0	0	
ror	rd, rs1, rs2	0	1	1	0	0	0	0	0	rs2					rs1		1	1	1	1	rd				0	0	1	0	0	0	0	
rori	rd, rs1, shamt	shamt	1	1	0	0	0	0	0	shamt					rs1		1	0	1	0	rd				0	0	1	0	0	0	0	
rorw	rd, rs1, shamt	shamt	1	1	0	0	0	0	0	shamt					rs1		1	0	1	0	rd				0	0	1	0	0	0	0	
ronw	rd, rs1, rs2	0	1	1	0	0	0	0	0	rs2					rs1		1	0	1	0	rd				0	1	1	0	0	0	0	
xnor	rd, rs1, rs2	0	1	0	0	0	0	0	0	rs2					rs1		1	0	0	0	rd				0	1	1	0	0	0	0	
zip	rd, rs	0	0	0	0	1	0	0	0	rs2					rs1		1	0	0	0	rd				0	1	1	0	0	0	0	
unzip	rd, rs	0	0	0	0	1	0	0	0	rs2					rs1		1	0	0	0	rd				0	1	1	0	0	0	0	
clmul	rd, rs1, rs2	0	0	0	0	1	0	0	0	rs2					rs1		1	0	0	0	rd				0	0	1	0	0	0	0	
clmulh	rd, rs1, rs2	0	0	0	0	1	0	0	0	rs2					rs1		1	0	0	0	rd				0	0	1	0	0	0	0	
xperm8	rd, rs1, rs2	0	0	1	0	0	0	0	0	rs2					rs1		1	0	0	0	rd				0	1	1	0	0	0	0	
xperm4	rd, rs1, rs2	0	0	1	0	0	0	0	0	rs2					rs1		0	1	0	0	rd				0	1	1	0	0	0	0	

Table 3.4: Detailed Decoding of Cryptography Instructions

- **Module Creation:** A dedicated module was developed for each cryptography extension. The design was guided by the computational details provided in the Sail model, as outlined in the RISC-V Cryptography Extensions manual.[3]
- **Co-Processor Design:** The primary module, which integrates with the main core, was designed to include the instruction decoder and all necessary components for the cryptography extensions. This co-processor module handles communication with the main core, managing offloaded instructions, validating inputs, and ensuring the correctness of outputs when ready. Additionally, it must efficiently interact with the main core, processing instructions even when the core is busy or issuing new commands.

Since some instruction types in these extensions are custom and involve immediate values such as `bs`, `rnum`, and `shmat`, which are not part of the standard `OP_IMM` instructions, the decoder must also extract these immediate values to ensure correct processing by the appropriate modules.

Additionally, certain instructions included in the cryptography extensions are already documented under other ISAs, such as the Bitmanipulation extension. These instructions do not need to be re-implemented in this co-processor.

3.5 Integration with CVA6 Core

The integration of cryptography extensions into the CVA6 core will be accomplished using the CORE-V eXtension InterFace (CVXIF). This interface allows for the seamless addition of custom coprocessors and ISA extensions without requiring modifications to the core's main RTL code.

The CVXIF is a versatile RISC-V extension interface that facilitates the implementation of custom or standardized instructions by extending the CPU with new capabilities. This integration is achieved without altering the CPU's original RTL design, making it an efficient and non-invasive method for enhancing processor functionality. **Key Features of the CVXIF:**

- **Low Latency:** Provides tightly integrated connections, ensuring minimal delay in instruction execution.
- **Read and Write Access:** Allows direct access to the CPU register file, enabling efficient data handling.
- **Minimal Instruction Encoding Requirements:** Supports extensions with minimal constraints on instruction encoding, promoting flexibility in design.

- **Dual Write-Back Support:** Enables instructions to write back to two registers simultaneously, enhancing performance for certain operations.
- **Dual Read Support:** Facilitates instructions that require reading from two registers at once.
- **Ternary Operation Support:** Capable of handling operations that involve three operands, increasing the complexity of possible instructions.

The cryptography accelerator will be connected to the CVA6 core via this interface, ensuring that the integration is smooth and maintains the integrity of the original core design.

3.6 Testing and Validation

A comprehensive testing strategy was employed to validate the implementation:

- **Functional Testing:** The correctness of each cryptography accelerator and its integration within the CVA6 core was verified. This testing was conducted using Verilator and VCS simulators.
- **Validation with Spike:** The Spike RISC-V simulator was used to validate the functionality and performance of the instructions. Spike served as a reference simulator to cross-check the execution results of each instruction.
- **Regression Testing:** The designed module was tested using 52 regression tests provided by RISC-V for the K extension.
- **Code Coverage Analysis:** Code coverage analysis was performed to ensure that all parts of the design were thoroughly tested.

These testing phases ensured the cryptography extensions were implemented correctly, delivered the expected security benefits, and did not adversely affect the overall functionality of the processor.

Chapter 4

Design and Implementation

4.1 CVA6 Core Overview

The CVA6 project aims to develop a family of high-quality, open-source RISC-V CPU cores suitable for both ASIC and FPGA implementations. These cores are written in SystemVerilog and are highly parameterizable, allowing configuration as either 32-bit (RV32) or 64-bit (RV64) cores with optional features such as floating-point support, MMU, PMP, and cache organization. CVA6 supports multiple operating systems, including Linux, FreeRTOS, and Zephyr, and is designed to be fully compliant with RISC-V specifications while allowing extensions through the CV-X-IF coprocessor interface.

CVA6 is an industrial evolution of the ARIANE core, originally developed by ETH Zürich and the University of Bologna, now maintained by the OpenHW Group. The core features a 6-stage pipeline and implements the RISC-V I, M, and C extensions, as well as the draft privilege extension, making it capable of running a full OS at efficient speeds. The CV32A6 and CV64A6 cores, which share the same SystemVerilog source code, are tailored to meet users' specific needs through a range of customizable parameters.

The overview of the CVA6 core is illustrated in Figure 4.1.

4.1.1 Pipeline Stages

The core features a 6-stage pipeline, consisting of the following stages:

- **Frontend (2 stages: PC Generation and Instruction Fetch):**
 - **PC Generation:** This stage is responsible for generating the next program counter (PC). All PCs are logically addressed. If there is a change in the logical to physical mapping, a `fence.vw` instruction is issued to flush

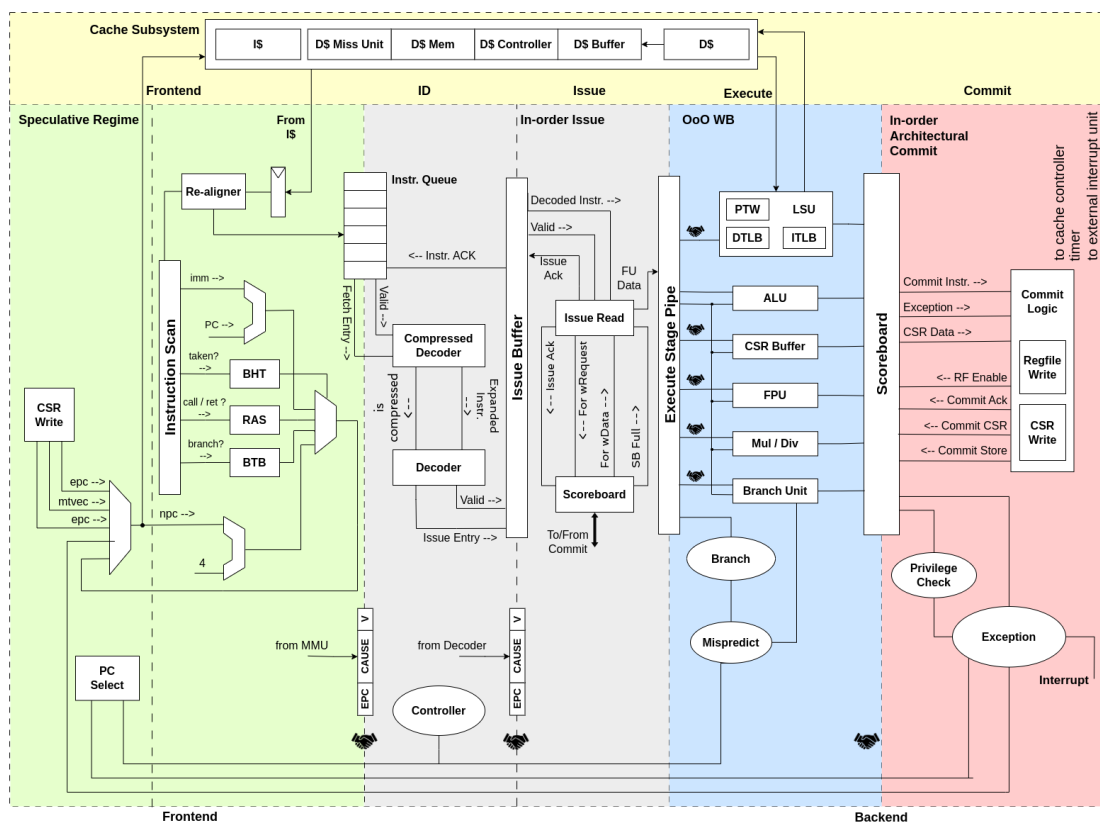


Figure 4.1: CVA6 core overview

the pipeline and TLBs. This stage includes speculation on the branch target address, branch prediction (whether the branch is taken or not), and contains both the branch target buffer (BTB) and branch history table (BHT).

- **Instruction Fetch (IF) Stage:** The IF stage receives information from the PC Generation stage, including branch prediction details (predicted branch, target address, taken/not taken), the current PC (word-aligned if it was a consecutive fetch), and the validity of the request. The IF stage then requests address translation from the MMU and controls the instruction cache interface (IS).
- **Instruction Decode (ID):** The ID stage is the first pipeline stage of the processor’s back-end. It decodes instructions from the data stream provided by the IF stage and passes them to the Issue stage. With the introduction of compressed or variable-length instructions, this stage also handles instruction re-alignment and decompression. Furthermore, branch instructions identified

at this stage are passed to the Issue stage.

- **Issue Stage:** The purpose of the Issue stage is to issue decoded instructions to the various functional units (FUs). It also tracks all issued instructions and manages the functional units' status, including receiving write-back data from the Execute stage. The Issue stage contains the CPU's register file and uses a data structure called a scoreboard to track instruction progress and register write-back locations. This stage manages the first, second, and fourth steps of instruction execution: issue, read operands, and write-back.
- **Execute Stage:** The Execute stage encapsulates all functional units (FUs), including the ALU, branch unit, load/store unit (LSU), CSR buffer, and multiply/divide unit. Each FU operates independently, with a valid signal for output data and a ready signal indicating the ability to accept new requests. The Execute stage processes instructions based on transaction IDs and returns results with the corresponding transaction ID and valid signal.
 - **Memory Management Unit (MMU):** The MMU is essential for virtual memory management and address translation in the CVA6 processor. It translates virtual addresses into physical addresses, providing memory protection, isolation, and efficient management. The MMU handles both instruction and data accesses and consists of key components like the Instruction TLB (ITLB), Data TLB (DTLB), optional Shared TLB, and the Page Table Walker (PTW).
- **Commit Stage:** The Commit stage finalizes instruction execution by updating the architectural state, including writing CSR registers, committing stores, and writing back data to the register file. This stage distinguishes between simple register write-back instructions and those requiring additional logic, such as store commits and freeing the CSR buffer upon instruction retirement.

4.1.2 PMP

The CVA6 core incorporates a Physical Memory Protection (PMP) unit, which offers both static and dynamic reconfigurability. The static configuration is determined by the top-level parameter `CVA6Cfg.NrPMPEntries`, while the dynamic configuration is managed through the Control and Status Registers (CSRs). The core supports up to 16 PMP entries.

Although all PMP CSRs are implemented, those corresponding to PMP entries with numbers equal to or greater than `CVA6Cfg.NrPMPEntries` are hardwired to zero. Upon reset, all PMP entries are initialized to zero.

When the L (Lock) bit is set, PMP restrictions are enforced even in Machine mode (M-mode).

The PMP grain is defined as 2^{G+2} , with CVA6 supporting only a granularity of 8 bytes (where $G = 1$). The PMP address length matches the processor's physical address length. Given that $G = 1$, the NA4 mode is not selectable.

Writes to `pmpaddr` are Write-Any-Read-Legal (WARL) and depend on the address mode. For naturally aligned power-of-two addressing mode (NAPOT), the bit is set to 1; for the top boundary of an arbitrary range (TOR), or when PMP is turned off (OFF), it is set to 0.

If, during a write to `pmpcfgX`, the Read (R) bit is 0 and the Write (W) bit is 1, the CSR will not be updated.

4.1.3 PMA

An underlying system can have multiple Physical Memory Attributes (PMA), and the CVA6 core supports three primary access properties:

- **Non-idempotent regions (I/O regions):** Due to the nature of a pipelined CPU architecture, the CPU may speculatively fetch (through branch prediction) and speculatively load (due to speculative execution). This behavior can cause issues if reads to a region are non-idempotent (i.e., they destroy state). Therefore, regions marked as non-idempotent will not be accessed speculatively. Common examples include UART registers or an interrupt claim register.
- **Executable regions (Main memory):** Regions marked as executable are considered code regions, allowing the core to fetch instructions from those regions.
- **Cacheable regions (Main memory):** Regions marked as cacheable are treated as data and instruction regions, enabling the core to fetch, load, and store data from those regions.

These regions are determined at instantiation-time and cannot be modified during runtime. The CVA6 core uses specific fields within the `cva6_cfg_t` configuration structure to statically define the PMA regions:

- `CVA6Cfg.NrNonIdempotentRules`: Number of active non-idempotent regions.
- `CVA6Cfg.NonIdempotentAddrBase`: Base address of the non-idempotent region.
- `CVA6Cfg.NonIdempotentLength`: Length of the non-idempotent region.
- `CVA6Cfg.NrExecuteRegionRules`: Number of active executable regions.

- `CVA6Cfg.ExecuteRegionAddrBase`: Base address of the executable region.
- `CVA6Cfg.ExecuteRegionLength`: Length of the executable region.
- `CVA6Cfg.NrCachedRegionRules`: Number of active cacheable regions.
- `CVA6Cfg.CachedRegionAddrBase`: Base address of the cacheable region.
- `CVA6Cfg.CachedRegionLength`: Length of the cacheable region.

Currently, the following RISC-V-defined PMAs are not supported by CVA6:

- **Coherence**: The stand-alone CVA6 does not support any coherence protocol, treating all memory accesses as non-coherent. In coherent systems like OpenPiton, it is the system's responsibility to define coherence attributes, where typically any cacheable region would also be coherent.
- **Atomicity**: Atomicity is managed by the underlying system, which determines whether atomicity is supported.
- **Reservability**: Similar to atomicity, reservability is handled by the underlying system.
- **Vacant Regions**: The determination of vacant regions is left to the underlying system. CVA6 does not check for vacant regions, and accessing such regions is expected to result in an AXI Decode Error or Slave Error.

4.2 Core-V eXtension InterFace (CV-X-IF)

The Core-V eXtension Interface (CV-X-IF) allows the CPU to be extended with custom or standardized instructions without altering the CPU's RTL (Register Transfer Level) design. Extensions are implemented in separate modules external to the CPU and integrated at the system level through this interface as a coprocessor.

The coprocessor operates like another functional unit so it is connected to the CVA6 in the execute stage.

4.2.1 Features

- **Low Latency Access:** The interface provides tightly integrated, low-latency read and write access to the CPU register file.
- **Custom Instruction Encoding:** CV-X-IF enables custom ALU type instructions, custom CSRs (Control and Status Registers), and other related instructions. Control-Transfer instructions (e.g., branches and jumps) are not supported.
- **Opcode Usage:** Unused opcodes by the CPU can be used for extensions, although it is recommended to avoid using opcodes reserved by RISC-V International.
- **Dual Write-Back:** Supports dual write-back instructions based on the `X_DUALWRITE` parameter for even-odd register pairs when `XLEN = 32`.
- **Dual Read:** Supports dual read operations per source operand based on the `X_DUALREAD` parameter, providing up to six 32-bit operands per instruction when `XLEN = 32`.
- **Ternary Operations:** Optionally supports ISA extensions for instructions using three source operands.
- **Instruction Speculation:** Indicates whether offloaded instructions should be committed or killed. (Unsupported in CVA6 yet)

4.2.2 Parameters

CV-X-IF defines two types of parameters.

1. **Coprocessor Parameters:** Configured for the coprocessor, though not all values may be supported by the CPU.
2. **System Parameters:** Determined by the configuration of both the CPU and the coprocessor, such as `X_ID_WIDTH` and `X_HARTID_WIDTH`.

Table 4.1 displays the supported CVXIF parameters in CVA6.

4.2.3 Interfaces

CV-X-IF consists of the following interfaces (as illustrated in Figure 4.2):

- **Compressed Interface:** Handles signaling for compressed instructions to be offloaded.

Name	Type/Range	Description
X_NUM_RS	int unsigned (2..3)	Number of register file read ports that can be used by the eXtension interface
X_ID_WIDTH	int unsigned (3..32)	Identification width for the eXtension interface
X_MEM_WIDTH	n/a (feature not supported)	Memory access width for loads/stores via the eXtension interface
X_RFR_WIDTH	int unsigned (32, 64)	Register file read access width for the eXtension interface
X_RFW_WIDTH	int unsigned (32, 64)	Register file write access width for the eXtension interface
X_MISA	logic [25:0]	MISA extensions implemented on the eXtension interface

Table 4.1: Supported CVXIF Parameters in CVA6

- **Issue Interface:** Manages request/response signaling for uncompressed instructions to be offloaded.
- **Register Interface:** Deals with signaling for GPRs (General Purpose Registers) and CSRs.
- **Commit Interface:** Signals control related to whether instructions should be committed or killed.
- **Result Interface:** Signals the completion of instruction execution and results.

Only the 3 mandatory interfaces (issue, commit, and result) have been implemented. Compressed interface, Memory Interface, and Memory result interface are not yet implemented in the CVA6.

4.2.4 Operating Principle

The CPU attempts to offload every instruction (compressed or non-compressed) that it does not recognize as valid. For compressed instructions, the coprocessor provides a matching uncompressed instruction, which is then offloaded via the issue interface.

As shown in Figure 4.3, the offloaded instruction's acceptance or rejection is decided by the coprocessor. If rejected, the CPU raises an illegal instruction exception. If accepted, the coprocessor handles the instruction, including managing register file operands through the register interface.

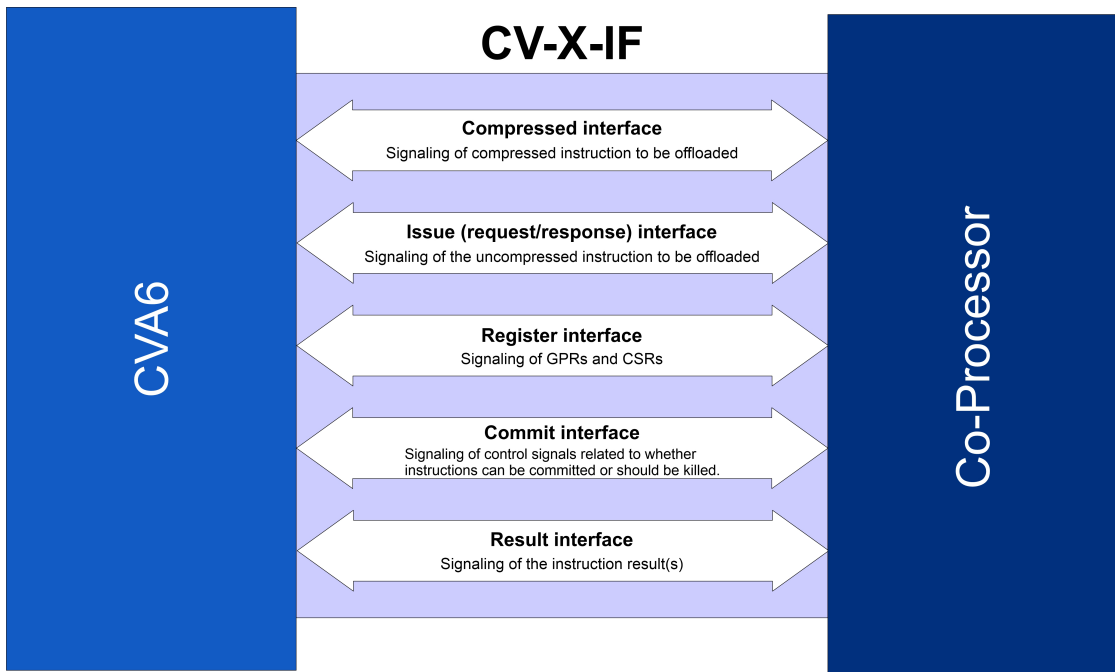


Figure 4.2: Interfaces in CV-X-IF

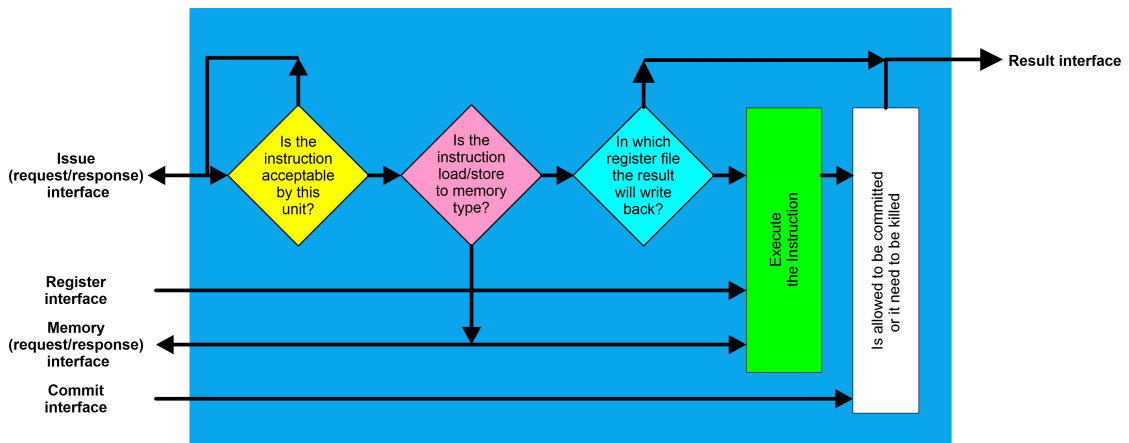


Figure 4.3: Operating Principle of CVXIF

Instructions offloaded to the coprocessor are speculative; the CPU may later decide to kill them based on certain conditions. The CPU uses the commit interface to inform the coprocessor about whether an instruction will be committed or killed.

The final result of an accepted offloaded instruction can be written back to the coprocessor or the CPU's register file, signaled through the result interface.

In the execution stage of the CVA6, a dedicated functional unit is implemented

to manage the CV-X-IF interfaces. The following sections describe the connections between these interfaces and the CVA6 core.

An explanation of the three interfaces used in CVA6 is provided below.

Issue Interface

- **Request Phase:** The operands are connected to the `issue_req.rs` signals. The scoreboard transaction ID is connected to the `issue_req.id` signal, ensuring that the scoreboard IDs and offloaded instruction IDs are linked together. This linkage allows the CVA6 to execute instructions out of order with the coprocessor, similar to other functional units within the core. The undecoded instruction is connected to the `issue_req.instruction` signal. The validity of the CV-X-IF functional unit is indicated by the `issue_req.valid` signal. All `issue_req.rs_valid` signals are set to 1, with the validity of source registers guaranteed by the `valid` signal transmitted from the issue stage.
- **Response Phase:** If the `issue_resp.accept` signal is asserted during a transaction (i.e., both `valid` and `ready` are set), the offloaded instruction is accepted by the coprocessor, leading to a result transaction. If `issue_resp.accept` is not asserted, the offloaded instruction is deemed illegal, and an illegal instruction exception will be raised as soon as no result transaction is written on the writeback bus.

Commit Interface

The `valid` signal of the commit interface is connected to the `valid` signal of the issue interface. Similarly, the `id` signal of the commit interface is connected to the `id` signal of the issue interface, which corresponds to the scoreboard ID. Notably, the killing of offloaded instructions is not supported in this implementation; therefore, all accepted offloaded instructions are committed to execution, and instruction termination is not possible.

Result Interface

- **Request Phase:** The `ready` signal of the result interface is always asserted, indicating that the CVA6 is always prepared to accept a result from the coprocessor for an accepted offloaded instruction.
- **Response Phase:** The result response is directly connected to the writeback bus of the CV-X-IF functional unit. The `valid` signal of the result interface is connected to the `valid` signal of the writeback bus. Additionally, the `id` signal of the result interface is connected to the scoreboard ID of the writeback bus. The write enable signal of the result interface is connected to a dedicated

CV-X-IF write enable (**WE**) signal in CVA6, which signals the scoreboard whether a writeback should occur in the CVA6 register file. The **exccode** and **exc** signals of the result interface are connected to the exception signals of the writeback bus. Notably, exceptions from the coprocessor do not populate the **tval** field in the exception signal of the writeback bus.

Three registers are incorporated to hold illegal instruction information in scenarios where a result transaction and a non-accepted issue transaction occur within the same cycle. In such cases, result transactions are prioritized and written to the writeback bus due to their association with an older offloaded instruction. Once the writeback bus is free, an illegal instruction exception will be raised using the information stored in these three registers.

4.3 Cryptography Accelerator

Figure 4.4 illustrates the block diagram of the CVA6 core, highlighting the placement of the CVXIF functional unit, which is specifically designed to execute cryptography extensions. As depicted, this functional unit is integrated within the execution stage. The flow of cryptography instructions is as follows:

- The instruction is fetched and placed in the instruction queue.
- The CVA6 decodes the instruction during the ID stage, identifies it as an offload instruction and determines that the CVXIF functional unit is required.
- The instruction is issued at the issue stage, with the associated information saved in the scoreboard.
- Data is transmitted through the CVXIF interfaces to the CVXIF functional unit.
- The functional unit processes the instruction, interacts with the coprocessor and retrieves the result.
- The result is stored in the scoreboard.
- The instruction is committed in order at the commit stage.

The scoreboard maintains the following information for each instruction:

- **PC**: The program counter (PC) of the instruction.
- **FU**: The functional unit to be used.

- **OP**: The operation to be performed within the functional unit.
- **RS1**: The address of the first source register.
- **RS2**: The address of the second source register.
- **RD**: The address of the destination register.
- **Result**: The result of the instruction. For unfinished instructions, this field also holds immediate value.
- **Valid**: A flag indicating whether the result is valid.
- **Use I Immediate**: A flag that indicates whether the immediate value should be used as operand B.
- **Use Z Immediate**: A flag that indicates whether the Z-Immediate (zimm) should be used as operand A.
- **Use PC**: A flag set when the PC should be used as operand A, typically in the context of exceptions.
- **Exception**: Indicates whether an exception has occurred.
- **Branch Predict**: Data related to branch prediction stored in the scoreboard.
- **Is Compressed**: A flag that signals whether the instruction is compressed. This information is crucial at the commit stage for determining the correct jump offset, such as +4 or +2.

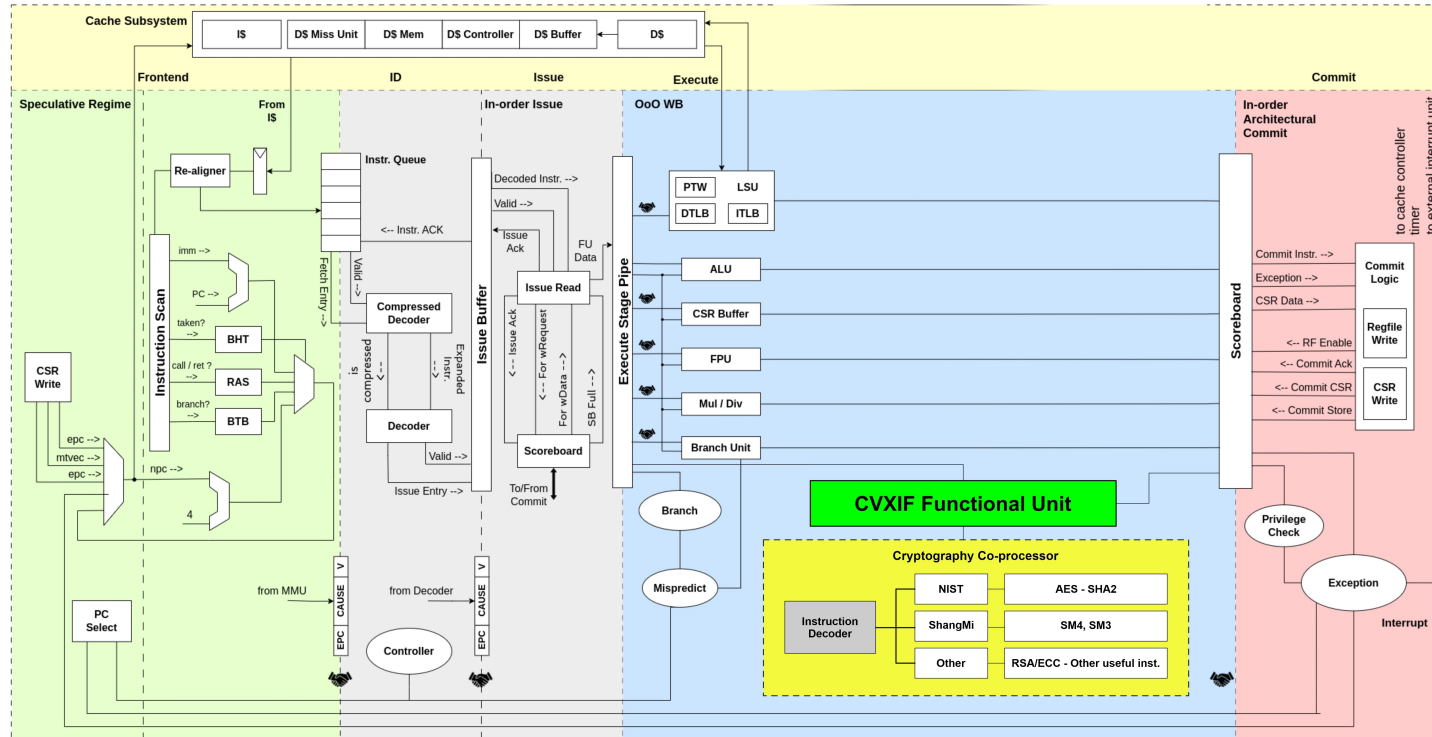


Figure 4.4: CVA6 + Cryptography Accelerator

Figure 4.5 presents the block diagram of the cryptography accelerator and its submodules, covering all instructions in the scalar cryptography extension Volume I [3].

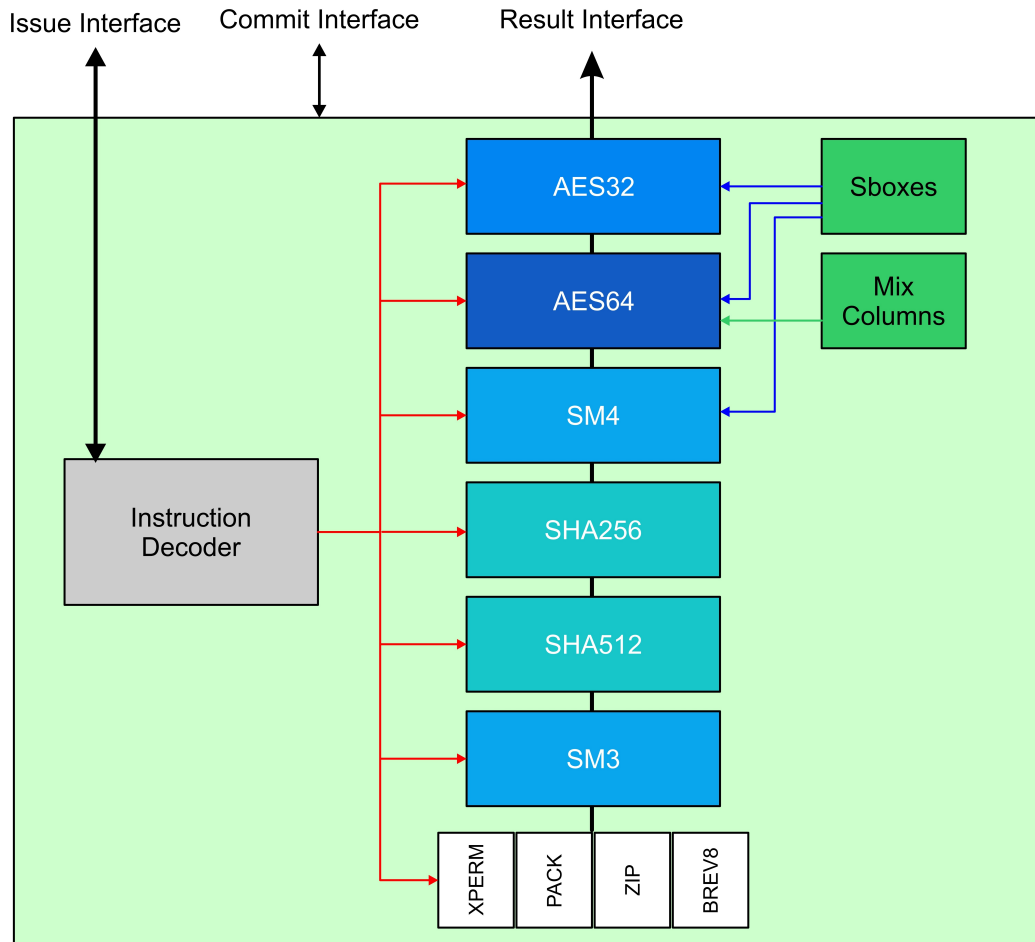


Figure 4.5: Cryptography Accelerator Block Diagram

As shown in the block diagram, instructions received from the issue interface are decoded, and the corresponding module processes the instruction to generate the result, which is then delivered to the result interface. The main core implements modules not represented in the figure through other extensions. The implementation of these modules is based on the Sail model provided in the manual. Since this project focuses on 64-bit operations, only 64-bit instructions are discussed in this section.

In this section, the instructions from the NIST and ShangMi suites are covered.

For a comprehensive overview of other cryptography instructions and details on their implementations, please refer to the Scalar Cryptography Manual [3].

4.3.1 AES Encryption/Decryption

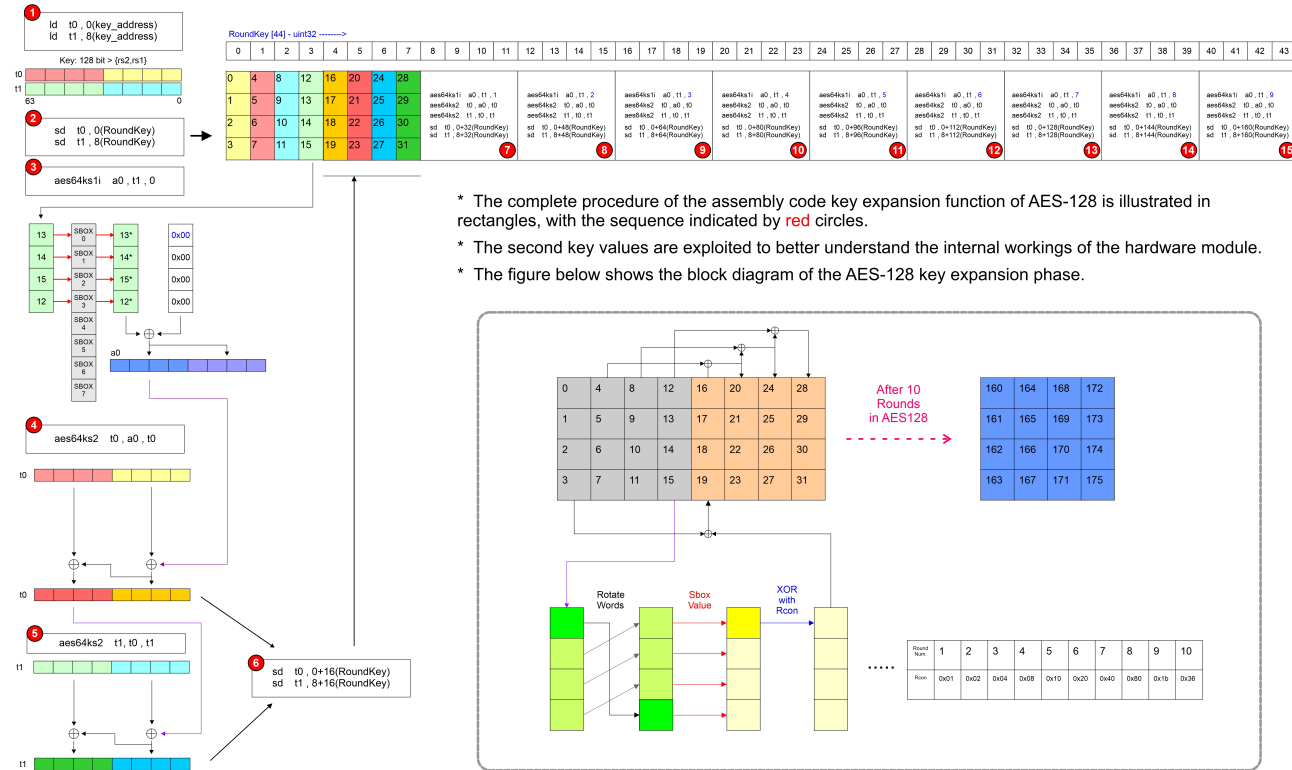
AES (Advanced Encryption Standard) is a symmetric encryption algorithm widely used for securing data. It operates on fixed-size blocks of data (typically 128 bits) and uses keys of 128, 192, or 256 bits. The encryption process involves multiple rounds, where each round includes steps like substitution (using S-boxes), permutation, mixing (using MixColumns), and adding the key (using XOR operations). Decryption is the reverse of encryption, applying inverse operations in reverse order to retrieve the original plaintext from the ciphertext. AES is known for its speed, security, and efficiency, making it a standard choice for encrypting sensitive data.

The encryption and decryption processes are performed in two phases: Key Expansion and Encrypt/Decrypt. These operations are completed after executing the required number of rounds for a set of instructions. Figures 4.7 and 4.6 illustrate these two phases and the necessary instructions for the encryption/decryption of a single block.

Within the cryptography accelerator, two submodules handle the calculation of S-boxes and Mix Columns. These operations are implemented using circuits rather than lookup tables (LUTs). For AES64, there are eight S-Box modules and two Mix Columns modules. The Mix Columns modules differ between the encryption and decryption phases, so there are a total of four. All AES instructions, as explained below, are executed within this module, with the result selected based on the decoded instruction. This implementation is a combinational circuit, meaning that the result is available in the same clock cycle as the issue stage.

The S-Box implementation is based on the techniques described in [5]. This reference presents new methods for reducing the depth of circuits used in cryptography applications while maintaining a relatively small gate count. Specifically, the AES S-Box circuit has a depth of 16 and uses only 128 gates. For the inverse S-Box, the depth is also 16, with 127 gates. Both the S-Box and its inverse share a common middle part consisting of 63 gates. In comparison, the best previous design for the AES S-Box had a depth of 22 and 148 gates.

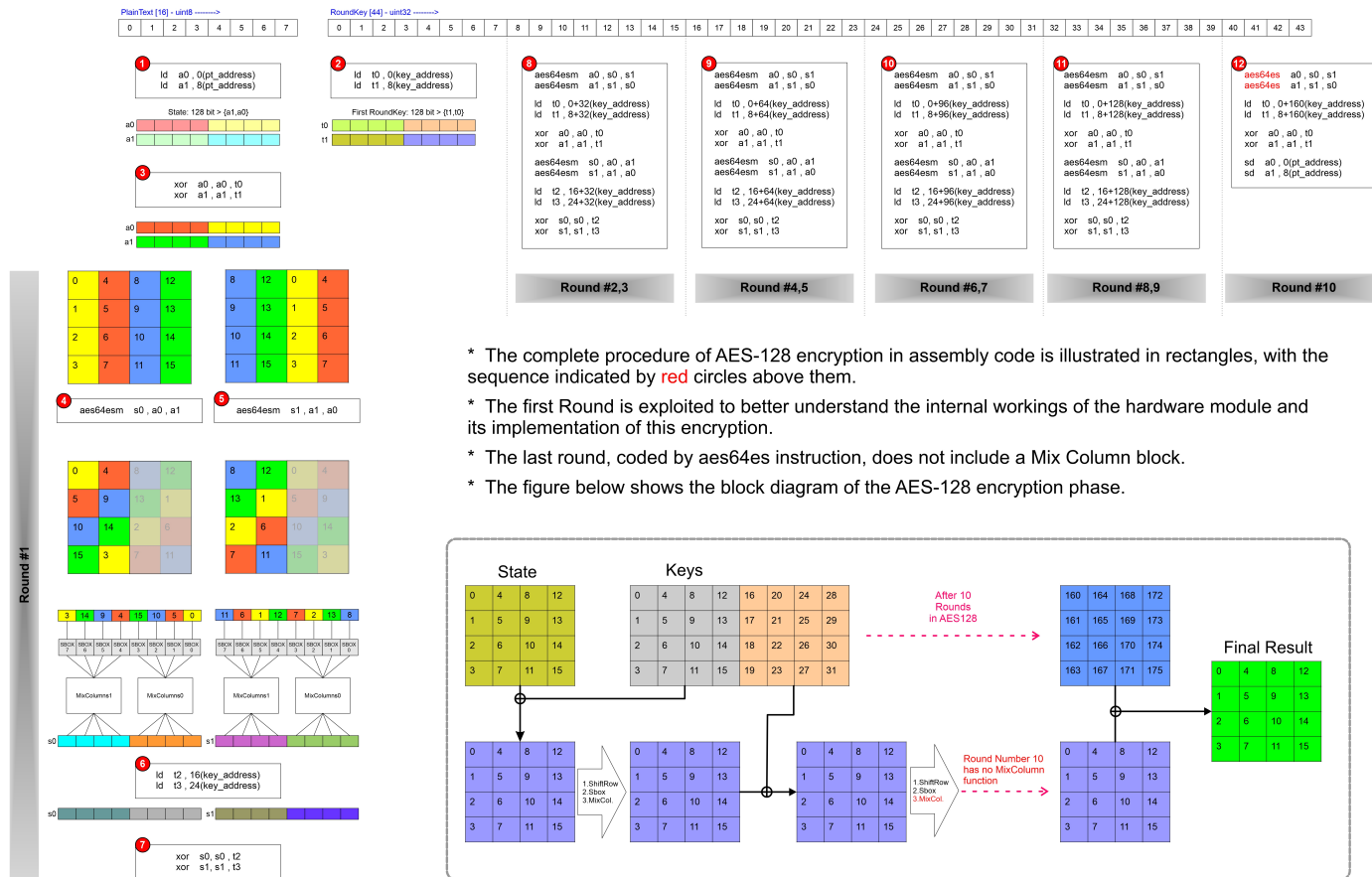
Key Expansion Function - AES128



- * The complete procedure of the assembly code key expansion function of AES-128 is illustrated in rectangles, with the sequence indicated by red circles.
- * The second key values are exploited to better understand the internal workings of the hardware module.
- * The figure below shows the block diagram of the AES-128 key expansion phase.

Figure 4.6: Key Expansion Phase of AES Encryption

Encryption Function - AES128



- * The complete procedure of AES-128 encryption in assembly code is illustrated in rectangles, with the sequence indicated by red circles above them.
- * The first Round is exploited to better understand the internal workings of the hardware module and its implementation of this encryption.
- * The last round, coded by aes64es instruction, does not include a Mix Column block.
- * The figure below shows the block diagram of the AES-128 encryption phase.

Figure 4.7: AES Encryption Phase for One Block

The Sail model for these instructions is provided below:

AES64DS: Uses the two 64-bit source registers to represent the entire AES state, and produces half of the next round output, applying the Inverse ShiftRows and SubBytes steps.

```

1 function clause execute (AES64DS(rs2, rs1, rd)) = {
2   let sr : bits(64) = aes_rv64_shiftrows_inv(X(rs2)[63..0],
3     X(rs1)[63..0]);
4   let wd : bits(64) = sr[63..0];
5   X(rd) = aes_apply_inv_sbox_to_each_byte(wd);
6   RETIRE_SUCCESS
7 }

```

AES64DSM: Uses the two 64-bit source registers to represent the entire AES state, and produces half of the next round output, applying the Inverse ShiftRows, SubBytes and MixColumns steps.

```

1 function clause execute (AES64DSM(rs2, rs1, rd)) = {
2   let sr : bits(64) = aes_rv64_shiftrows_inv(X(rs2)[63..0],
3     X(rs1)[63..0]);
4   let wd : bits(64) = sr[63..0];
5   let sb : bits(64) = aes_apply_inv_sbox_to_each_byte(wd);
6   X(rd) = aes_mixcolumn_inv(sb[63..32]) @ aes_mixcolumn_inv(
7     sb[31..0]);
8   RETIRE_SUCCESS
9 }

```

AES64ES: Uses the two 64-bit source registers to represent the entire AES state, and produces half of the next round output, applying the ShiftRows and SubBytes steps.

```

1 function clause execute (AES64ES(rs2, rs1, rd)) = {
2   let sr : bits(64) = aes_rv64_shiftrows_fwd(X(rs2)[63..0],
3     X(rs1)[63..0]);
4   let wd : bits(64) = sr[63..0];
5   X(rd) = aes_apply_fwd_sbox_to_each_byte(wd);
6   RETIRE_SUCCESS
7 }

```

AES64ESM: Uses the two 64-bit source registers to represent the entire AES state, and produces half of the next round output, applying the ShiftRows, SubBytes

and MixColumns steps.

```

1 function clause execute (AES64ESM(rs2, rs1, rd)) = {
2   let sr : bits(64) = aes_rv64_shiftrows_fwd(X(rs2)[63..0],
3     X(rs1)[63..0]);
4   let wd : bits(64) = sr[63..0];
5   let sb : bits(64) = aes_apply_fwd_sbox_to_each_byte(wd);
6   X(rd) = aes_mixcolumn_fwd(sb[63..32]) @ aes_mixcolumn_fwd(
7     sb[31..0]);
8   RETIRE_SUCCESS
9 }

```

AES64IM: The instruction applies the inverse MixColumns transformation to two columns of the state array, packed into a single 64-bit register.

```

1 function clause execute (AES64IM(rs1, rd)) = {
2   let w0 : bits(32) = aes_mixcolumn_inv(X(rs1)[31.. 0]);
3   let w1 : bits(32) = aes_mixcolumn_inv(X(rs1)[63..32]);
4   X(rd) = w1 @ w0;
5   RETIRE_SUCCESS
6 }

```

AES64KS1I: This instruction implements the rotation, SubBytes and Round Constant addition steps of the AES block cipher Key Schedule. Note that rnum must be in the range 0x0..0xA. The values 0xB..0xF are reserved.

```

1 function clause execute (AES64KS1I(rnum, rs1, rd)) = {
2   if(unsigned(rnum) > 10) then {
3     handle_illegal(); RETIRE_SUCCESS
4   } else {
5     let tmp1 : bits(32) = X(rs1)[63..32];
6     let rc : bits(32) = aes_decode_rcon(rnum); // round number
7       -> round constant
8
9     let tmp2 : bits(32) = if (rnum == 0xA) then tmp1 else
10       ror32(tmp1, 8);
11     let tmp3 : bits(32) = aes_subword_fwd(tmp2);
12     let result : bits(64) = (tmp3 ^ rc) @ (tmp3 ^ rc);
13     X(rd) = EXTZ(result);
14     RETIRE_SUCCESS
15   }
16 }

```

AES64KS2: This instruction implements the additional XOR'ing of key words as part of the AES block cipher Key Schedule.

```

1 function clause execute (AES64KS2(rs2, rs1, rd)) = {
2   let w0 : bits(32) = X(rs1)[63..32] ^ X(rs2)[31..0];
3   let w1 : bits(32) = X(rs1)[63..32] ^ X(rs2)[31..0] ^ X(rs2
4     ) [63..32];
5   X(rd) = w1 @ w0;
6   RETIRE_SUCCESS
  }
```

4.3.2 SHA-2 Hash Functions

The SHA-2 (Secure Hash Algorithm 2) family consists of cryptography hash functions designed to ensure data integrity. SHA-2 includes different hash functions with varying output sizes, notably SHA-256 and SHA-512, which produce hash values (also known as digests) of 256 and 512 bits, respectively. These functions process data in blocks (512 bits for SHA-256 and 1024 bits for SHA-512) and perform a series of operations including bitwise shifts, rotations, and modular additions. The final hash value, which is unique to the input data, can be used to verify data integrity or as a digital fingerprint in various security applications. SHA-2 is widely used in digital signatures, certificates, and password hashing due to its robustness and resistance to collision attacks.

The SHA-256 and SHA-512, are implemented using two distinct modules. These modules efficiently compute the necessary operations primarily through binary rotation, shifting, NOT, and XOR, as described in the Sail code snippets below.[3]

SHA-256

These instructions are supported for both RV32 and RV64 base architectures. For RV32, the entire XLEN source register is operated on. For RV64, the low 32 bits of the source register are operated on, and the result sign is extended to XLEN bits. Though named for SHA2-256, the instruction works for both the SHA2-224 and SHA2-256 parameterizations

SHA256SIG0: Implements the Sigma0 transformation function as used in the SHA2-256 hash function.

```

1 function clause execute (SHA256SIG0(rs1,rd)) = {
2   let inb : bits(32) = X(rs1)[31..0];
```

```
3   let result : bits(32) = ror32(inb, 7) ^ ror32(inb, 18) ^ (
4     inb >> 3);
5   X(rd) = EXTS(result);
6   RETIRE_SUCCESS
7 }
```

SHA256SIG1: Implements the Sigma1 transformation function as used in the SHA2-256 hash function.

```
1 function clause execute (SHA256SIG1(rs1,rd)) = {
2   let inb : bits(32) = X(rs1)[31..0];
3   let result : bits(32) = ror32(inb, 17) ^ ror32(inb, 19) ^
4     (inb >> 10);
5   X(rd) = EXTS(result);
6   RETIRE_SUCCESS
7 }
```

SHA256SUM0: Implements the Sum0 transformation function as used in the SHA2-256 hash function.

```
1 function clause execute (SHA256SUM0(rs1,rd)) = {
2   let inb : bits(32) = X(rs1)[31..0];
3   let result : bits(32) = ror32(inb, 2) ^ ror32(inb, 13) ^
4     ror32(inb, 22);
5   X(rd) = EXTS(result);
6   RETIRE_SUCCESS
7 }
```

SHA256SUM1: Implements the Sum1 transformation function as used in the SHA2-256 hash function.

```
1 function clause execute (SHA256SUM1(rs1,rd)) = {
2   let inb : bits(32) = X(rs1)[31..0];
3   let result : bits(32) = ror32(inb, 6) ^ ror32(inb, 11) ^
4     ror32(inb, 25);
5   X(rd) = EXTS(result);
6   RETIRE_SUCCESS
7 }
```

RV32: SHA-512

These instructions are implemented on RV32 only. Used to compute the transform of the SHA2-512 hash functions in conjunction with the other high/low half instruction. The transform is a 64-bit to 64-bit function, so the input and output are each represented by two 32-bit registers.

SHA512SIG0H: Implements the high half of the Sigma0 transformation, as used in the SHA2-512 hash function.

```

1 function clause execute (SHA512SIG0H(rs2, rs1, rd)) = {
2   X(rd) = EXTS((X(rs1) >> 1) ^ (X(rs1) >> 7) ^ (X(rs1) >> 8)
3     ^
4     (X(rs2) << 31) ^ (X(rs2) << 24) );
5   RETIRE_SUCCESS
6 }

```

SHA512SIG0L: Implements the low half of the Sigma0 transformation, as used in the SHA2-512 hash function.

```

1 function clause execute (SHA512SIG0L(rs2, rs1, rd)) = {
2   X(rd) = EXTS((X(rs1) >> 1) ^ (X(rs1) >> 7) ^ (X(rs1) >> 8)
3     ^
4     (X(rs2) << 31) ^ (X(rs2) << 25) ^ (X(rs2) << 24) );
5   RETIRE_SUCCESS
6 }

```

SHA512SIG1H: Implements the high half of the Sigma1 transformation, as used in the SHA2-512 hash function.

```

1 function clause execute (SHA512SIG1H(rs2, rs1, rd)) = {
2   X(rd) = EXTS((X(rs1) << 3) ^ (X(rs1) >> 6) ^ (X(rs1) >>
3     19) ^
4     (X(rs2) >> 29) ^ (X(rs2) << 13) );
5   RETIRE_SUCCESS
6 }

```

SHA512SIG1L: Implements the low half of the Sigma1 transformation, as used in the SHA2-512 hash function.

```

1 function clause execute (SHA512SIG1L(rs2, rs1, rd)) = {

```

```

2   X(rd) = EXTS((X(rs1) << 3) ^ (X(rs1) >> 6) ^ (X(rs1) >>
      19) ^
3   (X(rs2) >> 29) ^ (X(rs2) << 26) ^ (X(rs2) << 13) );
4   RETIRE_SUCCESS
5   }

```

SHA512SUM0R: Implements the Sum0 transformation, as used in the SHA2-512 hash function.

```

1   function clause execute (SHA512SUM0R(rs2, rs1, rd)) = {
2   X(rd) = EXTS((X(rs1) << 25) ^ (X(rs1) << 30) ^ (X(rs1) >>
      28) ^
3   (X(rs2) >> 7) ^ (X(rs2) >> 2) ^ (X(rs2) << 4) );
4   RETIRE_SUCCESS
5   }

```

SHA512SUM1R: Implements the Sum1 transformation, as used in the SHA2-512 hash function.

```

1   function clause execute (SHA512SUM1R(rs2, rs1, rd)) = {
2   X(rd) = EXTS((X(rs1) << 23) ^ (X(rs1) >> 14) ^ (X(rs1) >>
      18) ^
3   (X(rs2) >> 9) ^ (X(rs2) << 18) ^ (X(rs2) << 14) );
4   RETIRE_SUCCESS
5   }

```

RV64: SHA-512

These instructions are supported for the RV64 base architecture.

SHA512SIG0: Implements the Sigma0 transformation function as used in the SHA2-512 hash function.

```

1   function clause execute (SHA512SIG0(rs1, rd)) = {
2   X(rd) = ror64(X(rs1), 1) ^ ror64(X(rs1), 8) ^ (X(rs1) >>
      7);
3   RETIRE_SUCCESS
4   }

```

SHA512SIG1: Implements the Sigma1 transformation function as used in the SHA2-512 hash function.

```

1 function clause execute (SHA512SIG1(rs1, rd)) = {
2   X(rd) = ror64(X(rs1), 19) ^ ror64(X(rs1), 61) ^ (X(rs1) >>
3     6);
4   RETIRE_SUCCESS
}
```

SHA512SUM0: Implements the Sum0 transformation function as used in the SHA2-512 hash function.

```

1 function clause execute (SHA512SUM0(rs1, rd)) = {
2   X(rd) = ror64(X(rs1), 28) ^ ror64(X(rs1), 34) ^ ror64(X(
3     rs1), 39);
4   RETIRE_SUCCESS
}
```

SHA512SUM1: Implements the Sum1 transformation function as used in the SHA2-512 hash function

```

1 function clause execute (SHA512SUM1(rs1, rd)) = {
2   X(rd) = ror64(X(rs1), 14) ^ ror64(X(rs1), 18) ^ ror64(X(
3     rs1), 41);
4   RETIRE_SUCCESS
}
```

4.3.3 SM4 and SM3 Functions

SM4 and SM3 are cryptography algorithms standardized by the Chinese government for encryption and hashing, respectively.

SM4 is a symmetric block cipher algorithm used for encryption and decryption. It operates on 128-bit blocks and uses a 128-bit key. The algorithm involves 32 rounds of substitution and permutation operations, making it secure and efficient for various applications such as wireless communication and data encryption.

SM3 is a cryptography hash function similar in structure to SHA-2. It generates a 256-bit hash value from input data of any length. SM3 is designed to provide data integrity and authentication, ensuring that even a small change in the input data produces a significantly different hash value. It's widely used in digital signatures and other security protocols within China.

Similar to the AES algorithm, the SM4 S-Box is computed using a combinational circuit, as described in the same paper referenced in the AES section [5].

The Sail model for these functions is detailed in the [3], which outlines how these cryptography algorithms are implemented in a formalized manner.

These instructions are supported for the RV32 and RV64 base architectures.

SM3P0: Implements the P0 transformation function as used in the SM3 hash function.

```

1 function clause execute (SM3P0(rs1, rd)) = {
2   let r1 : bits(32) = X(rs1)[31..0];
3   let result : bits(32) = r1 ^ rol32(r1, 9) ^ rol32(r1, 17);
4   X(rd) = EXTS(result);
5   RETIRE_SUCCESS
6 }

```

SM3P1: Implements the P1 transformation function as used in the SM3 hash function.

```

1 function clause execute (SM3P1(rs1, rd)) = {
2   let r1 : bits(32) = X(rs1)[31..0];
3   let result : bits(32) = r1 ^ rol32(r1, 15) ^ rol32(r1, 23)
4   ;
5   X(rd) = EXTS(result);
6   RETIRE_SUCCESS
7 }

```

SM4ED: Accelerates the block encrypt/decrypt operation of the SM4 block cipher. Implements a T-tables in hardware style approach to accelerating the SM4 round function. A byte is extracted from rs2 based on bs, to which the SBox and linear layer transforms are applied, before the result is XOR'd with rs1 and written back to rd. On RV64, the 32-bit result is sign extended to XLEN bits.

```

1 function clause execute (SM4ED (bs,rs2,rs1,rd)) = {
2   let shamt : bits(5) = bs @ 0b000; // shamt = bs*8 //
3   let sb_in : bits(8) = (X(rs2)[31..0] >> shamt)[7..0];
4   let x      : bits(32) = 0x000000 @ sm4_sbox(sb_in);
5   let y      : bits(32) = x ^ (x << 8) ^ (x << 2) ^
6     (x << 18) ^ ((x & 0x0000003F)
7     << 26) ^
8     ((x & 0x000000C0) << 10);
9   let z      : bits(32) = rol32(y, unsigned(shamt));
10  let result : bits(32) = z ^ X(rs1)[31..0];

```



```

10 X(rd) = EXTS(result);
11 RETIRE_SUCCESS
12 }

```

SM4KS: Accelerates the Key Schedule operation of the SM4 block cipher. Implements a T-tables in hardware style approach to accelerating the SM4 Key Schedule. A byte is extracted from rs2 based on bs, to which the SBox and linear layer transforms are applied, before the result is XOR'd with rs1 and written back to rd. On RV64, the 32-bit result is sign extended to XLEN bits.

```

1 function clause execute (SM4KS (bs,rs2,rs1,rd)) = {
2   let shamt : bits(5) = (bs @ 0b000); // shamt = bs*8 //
3   let sb_in : bits(8) = (X(rs2)[31..0] >> shamt)[7..0];
4   let x      : bits(32) = 0x000000 @ sm4_sbox(sb_in);
5   let y      : bits(32) = x ^ ((x & 0x00000007) << 29) ^ ((x
6     & 0x000000FE) << 7) ^
7     ((x & 0x00000001) << 23) ^ ((x
8     & 0x000000F8) << 13) ;
9   let z      : bits(32) = rol32(y, unsigned(shamt));
10  let result: bits(32) = z ^ X(rs1)[31..0];
11  X(rd) = EXTS(result);
12  RETIRE_SUCCESS
13 }

```

Chapter 5

Testing and Validation

5.1 Functional Testing of Modules

Functional testing ensures that each module performs its intended operations correctly. Test suites developed by the RISC-V community in the K extension are used to verify the functionality of each instruction. These test suites consist of several assembly programs where specific instructions are executed multiple times with varying input values. In the initial phase, each instruction is tested manually by examining the output waveforms to evaluate the results of each module. Figure 5.1 shows an example of the output for instructions of the SHA2 family. This simulation is performed using the VCS and Verilator simulators. Each instruction is tested individually with a single input to debug the hardware and connections between submodules and the main core.

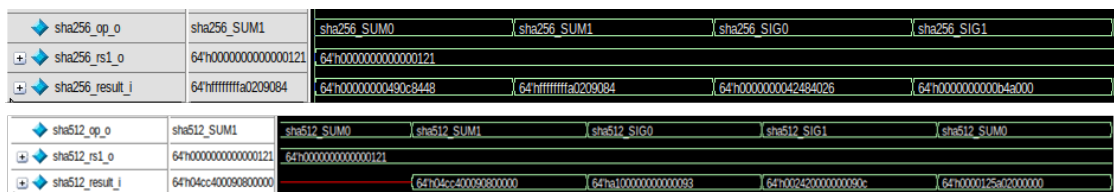


Figure 5.1: Waveforms of the SHA2 family results

In the second phase, Spike, the RISC-V ISA simulator, is used to automatically validate the results of each instruction with multiple inputs. Results from Spike and the core simulator are logged, and after simulation, the log data are compared to verify register values after executing each line.

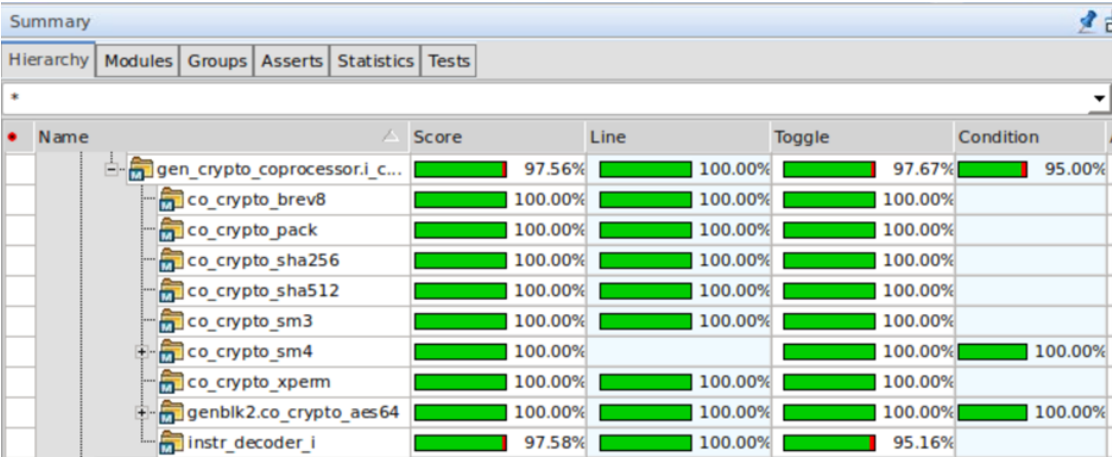
5.2 Regression Test and Code Coverage Analysis

Regression testing was conducted to ensure that integrating the cryptography extensions did not introduce any regressions or unintended behaviors in the CVA6 core. This involved running a comprehensive suite of tests where the core executes all tests sequentially without terminating the simulation. After executing all 52 tests, the simulation results for each file are represented in Table 5.1. The table displays the number of assembly instructions in each file, simulation time, and execution details.

By running all instructions without terminating the simulation, code coverage metrics for the cryptography accelerator were obtained. Code coverage analysis, performed using VCS and Verdi, included:

- **Statement Coverage:** Ensuring that every statement in the HDL code was executed at least once.
- **Path Coverage:** Ensuring that all possible paths through the code were exercised.

High code coverage percentages indicate a robust test suite, providing confidence in the correctness and completeness of the hardware design. Figure ?? shows the code coverage results. The identified gaps were due to unused signals from the main core to the co-processor and incomplete coverage of all registered files.



Name	Score	Line	Toggle	Condition
gen_crypto_coprocessor.i_c...	97.56%	100.00%	97.67%	95.00%
co_crypto_brev8	100.00%	100.00%	100.00%	
co_crypto_pack	100.00%	100.00%	100.00%	
co_crypto_sha256	100.00%	100.00%	100.00%	
co_crypto_sha512	100.00%	100.00%	100.00%	
co_crypto_sm3	100.00%	100.00%	100.00%	
co_crypto_sm4	100.00%		100.00%	100.00%
co_crypto_xpem	100.00%	100.00%	100.00%	
genblk2.co_crypto_aes64	100.00%	100.00%	100.00%	100.00%
instr_decoder_i	97.58%	100.00%	95.16%	

Figure 5.2: Code Coverage Results

Test File	Simulation Time (ns)	#Clock_Cycle	CPU Time (s)	Assembly Code (lines)
aes64ds-01.S	11478.5	7582	5.58	1288
aes64ds-rwp1.S	5607.5	3668	5.13	572
aes64dsm-01.S	11478.5	7582	4.65	1288
aes64dsm-rwp1.S	5607.5	3668	3.71	572
aes64es-01.S	11478.5	7582	5.53	1288
aes64esm-01.S	11478.5	7582	5.04	1288
aes64esm-rwp1.S	5607.5	3668	3.3	572
aes64es-rwp1.S	5607.5	3668	3.87	572
aes64im-01.S	9270.5	6110	6.42	1020
aes64im-rwp1.S	18408.5	12202	6.87	2044
aes64im-rwp2.S	4251.5	2764	4.13	382
aes64ks1i-01.S	10137.5	6688	5.91	1123
aes64ks2-01.S	17475.5	11580	10.33	1992
brev8-01.S	11925.5	7880	6.54	1344
pack-01.S	75117.5	50008	19.01	8956
packh-01.S	75117.5	50008	51.45	8956
packw-01.S	75117.5	50008	50.3	8956
sha256sig0-01.S	9270.5	6110	6.33	1020
sha256sig0-rwp1.S	18408.5	12202	7.61	2044
sha256sig0-rwp2.S	4251.5	2764	3.58	382
sha256sig1-01.S	9270.5	6110	6.1	1020
sha256sig1-rwp1.S	18408.5	12202	7.29	2044
sha256sig1-rwp2.S	4251.5	2764	3.36	382
sha256sum0-01.S	9270.5	6110	6.28	1020
sha256sum0-rwp1.S	18408.5	12202	7.66	2044
sha256sum0-rwp2.S	4251.5	2764	4.8	382
sha256sum1-01.S	9270.5	6110	6.79	1020
sha256sum1-rwp1.S	18408.5	12202	8.91	2044
sha256sum1-rwp2.S	4251.5	2764	3.98	382
sha512sig0-01.S	9270.5	6110	5.74	1020
sha512sig0-rwp1.S	18408.5	12202	8.31	2044
sha512sig0-rwp2.S	4251.5	2764	4.04	382
sha512sig1-01.S	9270.5	6110	6.46	1020
sha512sig1-rwp1.S	18408.5	12202	6.93	2044
sha512sig1-rwp2.S	4251.5	2764	4.07	382
sha512sum0-01.S	9270.5	6110	5.93	1020
sha512sum0-rwp1.S	18408.5	12202	7.25	2044
sha512sum0-rwp2.S	4251.5	2764	3.92	382
sha512sum1-01.S	9270.5	6110	6.02	1020
sha512sum1-rwp1.S	18408.5	12202	7.93	2044
sha512sum1-rwp2.S	4251.5	2764	3.36	382
sm3p0-01.S	9270.5	6110	6.43	1020
sm3p0-rwp1.S	18408.5	12202	8.25	2044
sm3p0-rwp2.S	4251.5	2764	4.98	382
sm3p1-rwp1.S	18408.5	12202	6.75	2044
sm3p1-rwp2.S	4251.5	2764	3.88	382
sm4ed-01.S	45039.5	29956	24.81	5320
sm4ed-rwp1.S	12654.5	8366	5.89	1431
sm4ks-01.S	45039.5	29956	23.26	5320
sm4ks-rwp1.S	12654.5	8366	6.97	1431
xperm4-01.S	78480.5	52250	54.99	9350
xperm8-01.S	57084.5	37986	32.29	6791

Table 5.1: Simulation Results of Regression Tests

5.3 Summary

In summary, the testing and validation of the cryptography modules involved rigorous functional and regression testing to ensure accurate operation and integration. Functional testing was performed both manually and automatically using different simulators to verify instruction functionality. Regression testing confirmed that no new issues were introduced by the cryptography extensions. Code coverage analysis further validated that the test suite thoroughly exercised the design, though some gaps were noted and addressed. These comprehensive testing efforts ensure the reliability and correctness of the cryptography modules in the CVA6 core.

Chapter 6

Results and Discussion

The results for area and power consumption were extracted using Synopsys tools with the *NangateOpenCellLibrary* technology library. For power analysis, an AES block encryption and decryption was executed, as described in the *Performance Analysis* section.

The VCD (Value Change Dump) file for these programs was generated using Verilator and served as the switching activity input for the power report. In the following sections, the *Area Report* and *Power Report* present the respective results.

The critical path of the design is inside the CVA6 core and is not related to the added co-processor. Since these paths are independent of the added modules and are based on the main core itself, a detailed timing report is not covered in this thesis. However, the final achievable synthesis using this technology library yields a clock period of 3.6 ns, corresponding to a frequency of approximately 277.78 MHz.

6.1 Area Report

This section provides a detailed comparison of the area utilization between the CVA6 core and the Cryptography accelerator. The table 6.1 summarizes the key metrics and areas for both components, expressed in square micrometers (μm^2).

The total combinational area in CVA6 is **233,009.3510** μm^2 , whereas in the Cryptography accelerator, it is significantly smaller, at **8,905.9459** μm^2 . The non-combinational area follows a similar pattern: CVA6 occupies **128,095.2301** μm^2 , while the Cryptography accelerator occupies only **194.1800** μm^2 .

When the Cryptography accelerator is integrated with the CVA6 core, there is an overall increase in area by **2.52%**. This suggests that while the Cryptography accelerator adds functionality, it does so with minimal impact on the overall area.

Metric	CVA6	Cryptography Accelerator
Number of ports	2451	504
Number of nets	242354	7295
Number of cells	225408	7264
Number of combinational cells	200805	7187
Number of sequential cells	24106	73
Number of buf/inv	33844	1145
Number of references	82	32
Area (μm^2)	CVA6	Cryptography Accelerator
Combinational Area	233,009.3510	8,905.9459
Buf/Inv Area	21,014.2659	618.1840
Noncombinational Area	128,095.2301	194.1800
Absolute Total Area	361,104.5811	9,100.1259

Table 6.1: Comparison of Area Metrics between CVA6 and Cryptography Accelerator

6.2 Power Report

The power report generated by the Synopsys Compiler is divided into two main components: dynamic power and static power. The total power consumption (P_{total}) is calculated using the following formula:

$$P_{\text{total}} = P_{\text{dynamic}} + P_{\text{static}}$$

Static Power

Static power is the power dissipated when the circuit is not switching, which is primarily caused by leakage currents. The total static power is given by the following formula:

$$P_{\text{static}} = \sum_{\forall \text{cell}_i} \text{leakage_power}(\text{cell}_i)$$

The leakage power consumption for each standard cell is specified in the technology library.

Dynamic Power

Dynamic power is the power dissipated when the circuit is active, i.e., when the voltage on a net changes due to external stimuli. Dynamic power is composed of two components: switching power and internal power.

Switching Power

Switching power is the power dissipated due to the charging and discharging of the capacitive load at the output. The load capacitance is contributed by both the net capacitance (which can be estimated or back-annotated) and the input pin capacitances of the fanout cells. Switching power depends on:

- Total capacitive load,
- Rate of logic transitions,
- The supply voltage V_{dd} .

The total switching power is calculated as:

$$P_{\text{switching}} = \sum_{\forall \text{net}_i} \text{switching_power}(\text{net}_i)$$

The switching power for a given net can be computed using the following formula, derived from solving a simple RC circuit:

$$\text{switching_power}(\text{net}_i) = 0.5 \times V_{dd}^2 \times C_{\text{load}}(\text{net}_i) \times \text{toggle_rate}(\text{net}_i)$$

Here:

- $C_{\text{load}}(\text{net}_i)$ is the load capacitance on the net,
- $\text{toggle_rate}(\text{net}_i)$ is the number of transitions per unit of time.

Internal Power

Internal power is the power dissipated within the boundaries of a cell. This includes power dissipation due to:

- The charging and discharging of capacitances internal to the cell,
- A short-circuit current flowing momentarily between the pull-up and pull-down networks during switching.

The total internal power is computed as:

$$P_{\text{internal}} = \sum_{\forall \text{cell}_i} \text{internal_power}(\text{cell}_i)$$

Power Reports

Table 6.2 shows the power report of the CVA6 core for different programs, while Table 6.3 presents the power consumption of the cryptography accelerator separately. In the case where the cryptography accelerator is used, the total power consumption should be the sum of both the CVA6 and the accelerator. This combined total can then be compared with the power consumption when the cryptography accelerator is not used.

Power Group	Internal Power (W)	Switching Power (mW)	Leakage Power (mW)	Total Power (W)	Execution Time(cycles)
With Accelerator Encryption	309.0295	502.5311	7.5093	309.5320	4,274
With Accelerator Decryption	309.0252	501.8715	7.5094	309.5271	4,460
Without Acc. Encryption	309.0554	519.8745	7.5098	309.5753	66,960
Without Acc. Decryption	309.0372	522.9204	7.5095	309.5601	405,544

Table 6.2: CVA6 Power Report (With and Without Accelerator)

Power Group	Internal Power (mW)	Switching Power (mW)	Leakage Power (uW)	Total Power (mW)	Execution Time(cycles)
Encryption	9.3427	7.6979	210.4243	17.0406	4,274
Decryption	11.9716	9.6859	210.4214	21.6574	4,460

Table 6.3: Cryptography Accelerator Power Report (Encryption and Decryption)

The total power consumed by the core is the sum of the power consumed by the CVA6 core and the cryptography accelerator:

$$P_{\text{total}} = P_{\text{CVA6}} + P_{\text{Accelerator}}$$

When the cryptography accelerator is not used, only the power consumption of the CVA6 core is considered:

$$P_{\text{total}} = P_{\text{CVA6}}$$

For encryption, the CVA6 core alone consumes approximately 309.58 W, while adding the cryptography accelerator increases the total power to 309.60 W (CVA6 + 17.04 mW from the accelerator). This represents an increase of approximately 0.007% in total power consumption.

Similarly, for decryption, the CVA6 core alone consumes 309.56 W, and adding the cryptography accelerator brings the total to 309.58 W (CVA6 + 21.66 mW from the accelerator). This results in a 0.01% increase in total power.

This comparison highlights how much power is related to the added module in case of using the accelerator. However, without the accelerator, the program's execution time increases, resulting in higher internal power and leakage power, compared to when the accelerator is used, and the program finishes more quickly. Consequently, longer execution times lead to greater power consumption.

It can be observed that the CVA6 power in encryption without the accelerator is 309.5753 W and with the accelerator is 309.5320 W, while in the decryption phase, the power without the accelerator is 309.5601 W and with the accelerator is 309.5271 W.

At the end, comparing the total power of the whole core:

- Encryption: without accelerator = 309.5753 W, with accelerator = 309.60 W.
- Decryption: without accelerator = 309.5601 W, with accelerator = 309.58 W.

Analysis: The use of the cryptography accelerator results in a slightly higher total power, but due to the reduced execution time, it also lowers internal and leakage power, making the overall power efficiency more favorable.

6.3 Performance Analysis

For the performance analysis, two types of C programs were evaluated to assess the performance of AES, which is a critical component in this implementation. The core developed and executed four C programs, with results checked for encryption and decryption. Specifically, two C programs were used for encryption and two for decryption, with each developed to run both with and without cryptography extensions.

To measure performance, the number of instructions executed and the execution time were recorded for a single block of data during the simulation. The results are summarized in Table 6.4.

As shown in the table, the use of the cryptography accelerator significantly improves both the number of instructions executed and the execution time.

- **Encryption:** Without the accelerator, the process required 27,465 instructions and took 66,960 cycles to complete. With the accelerator, the instruction count was reduced to just 158, and the execution time dropped dramatically to 4,274 cycles. This represents a performance improvement of approximately 94% in execution time.

Operation	Accelerator	Number of Instructions	Execution Time (cycles)
Encryption	Without Accelerator	27,465	66,960
	With Accelerator	158	4,274
Decryption	Without Accelerator	205,439	405,544
	With Accelerator	205	4,460

Table 6.4: Performance Analysis of AES with/without Cryptography Extensions

- **Decryption:** Without the accelerator, the process required 205,439 instructions and took 405,544 cycles. With the accelerator, the instruction count was reduced to 205, with the execution time reduced to 4,460 cycles, indicating a performance improvement of approximately 98% in execution time.

The "Number of Instructions" refers to the cumulative count of assembly instructions executed from the initiation of the KeyExpansion phase until the conclusion of the encryption/decryption function. The execution time represents the total cycles measured during the Verilator simulation of the compiled C program. This substantial reduction in both instruction count and execution time highlights the efficiency and effectiveness of the cryptography extensions in optimizing the AES operations.

6.4 Discussion

The successful integration of cryptography extensions into the CVA6 core demonstrates the feasibility and benefits of incorporating hardware-level security features into modern processors. The outcomes of this project offer valuable insights into various aspects of processor design, particularly in the context of balancing performance, security, and resource efficiency.

6.4.1 Design Trade-offs

The integration of cryptography extensions required careful consideration of several design trade-offs. While the inclusion of dedicated cryptography hardware increased the core's resource utilization, the performance gains—especially in terms of reduced instruction count and execution time—made these trade-offs worthwhile. The modular nature of the CVA6 core proved advantageous, allowing for seamless integration of new features without significant disruption to the core architecture. The enhancements in encryption and decryption performance, as evidenced by the reduction in execution cycles, underscore the efficiency of the design despite the added complexity.

6.4.2 Security Implications

The addition of hardware-based cryptography extensions substantially bolstered the security posture of the CVA6 core. By offloading critical cryptographic operations from software to dedicated hardware, the design effectively mitigated the risks associated with software-based vulnerabilities and side-channel attacks. This hardware-centric approach not only enhances data integrity and confidentiality but also provides a more secure execution environment for applications demanding high levels of security. The shift towards hardware-based security mechanisms aligns with the growing need for robust and resilient systems in an increasingly interconnected world.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis has successfully demonstrated the design and implementation of a RISC-V processor integrated with advanced security features, addressing contemporary cybersecurity challenges. The primary objective was to incorporate the newly ratified RISC-V Cryptography Extensions into a 64-bit core, significantly enhancing the processor's capability to execute cryptography tasks both efficiently and securely.

The key accomplishments of this work include:

- **Core Selection and Integration:** The project commenced with a thorough evaluation of existing 64-bit RISC-V cores, ultimately selecting the CVA6 core as the optimal platform for integration. This core, adhering to the 64-bit RISC-V instruction set, provided a robust foundation for the addition of cryptography functionalities.
- **Cryptography Accelerator Development:** A cryptography accelerator was meticulously designed and integrated into the CVA6 core. The development process involved rigorous functional testing, spike simulation validation, and comprehensive regression testing to ensure the correct and efficient operation of the cryptography instructions.
- **Performance Improvements:** The integration of the cryptography accelerator led to significant performance improvements, particularly in the implementation of the AES algorithm. The results demonstrated an approximate 15-fold increase in speed and a reduction in code size, validating the effectiveness of the hardware-level enhancements.
- **Security Enhancements:** By isolating critical cryptography functions from the main processor and ensuring constant-time execution of algorithms, the

implementation significantly reduced vulnerabilities to attacks such as side-channel threats. These hardware security measures provide a robust defense against various cyber threats.

In conclusion, this thesis contributes valuable insights into secure processor design, particularly within the context of the open-source RISC-V architecture. The enhancements achieved not only improve the security capabilities of the CVA6 core but also lay the groundwork for future research and development in hardware-based security mechanisms.

7.2 Future Work

The successful integration of cryptography extensions into the CVA6 core opens several avenues for future research and development. Potential areas of exploration include:

- **Advanced Cryptography Algorithms:** Extending the hardware support to include more complex algorithms such as elliptic curve cryptography (ECC) and post-quantum cryptography schemes. This would further enhance the security features of the processor, making it suitable for a wider range of applications.
- **Optimized Hardware Architectures:** Investigating further optimizations in hardware design to improve performance and reduce resource overhead. This could involve refining the accelerator design or exploring alternative hardware architectures that balance performance, area, and power consumption more effectively.
- **Integration with Other Security Features:** Combining cryptography extensions with other hardware-based security mechanisms, such as secure boot and trusted execution environments (TEEs), to create a more comprehensive and secure processing environment. This integration could provide a more holistic approach to security in embedded systems and other applications.
- **Scalability and Flexibility:** Exploring the scalability of the cryptography accelerator to support a broader range of applications, from low-power embedded systems to high-performance computing environments. Additionally, investigating ways to make the accelerator more flexible, allowing for dynamic reconfiguration based on the security requirements of different applications.
- **Real-world Applications and Benchmarks:** Evaluating the performance and security of the enhanced CVA6 core in real-world applications and benchmarks. This would involve deploying the core in practical scenarios to measure

its effectiveness in a variety of use cases, providing valuable feedback for further improvements.

The work presented in this thesis establishes a strong foundation for future developments in secure processor design, particularly in the context of the open-source RISC-V architecture. As cybersecurity threats continue to evolve, the need for robust, hardware-based security measures will only grow, making this an exciting and critical area of ongoing research.

Bibliography

- [1] *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*. RISC-V Foundation. Dec. 2019 (cit. on p. 3).
- [2] *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. RISC-V Foundation. Apr. 2024 (cit. on pp. 3, 4).
- [3] *RISC-V Cryptography Extensions Volume I: Scalar & Entropy Source Instructions*. RISC-V Foundation. Feb. 2022 (cit. on pp. 5, 12, 27, 28, 33, 38).
- [4] F. Zaruba and L. Benini. «The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (Nov. 2019), pp. 2629–2640. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2019.2926114 (cit. on p. 5).
- [5] Joan Boyar and René Peralta. «A Small Depth-16 Circuit for the AES S-Box». In: *Information Security and Privacy Research*. Ed. by Dimitris Gritzalis, Steven Furnell, and Marianthi Theoharidou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 287–298. ISBN: 978-3-642-30436-1 (cit. on pp. 28, 37).