POLITECNICO DI TORINO

Master's Degree in Computer Engineering - Embedded Systems

Master's Degree Thesis

Scalar Cryptography Extensions for STxP5

Supervisors

Prof. Guido MASERA

Candidate

Marco CHIARLE

October 2024

Abstract

The fast evolution of digital technology has brought in an era where security is not negligible, especially in the world of the embedded systems. As Cyber threats are always more sophisticated, the demand for robust cryptography solutions is increasingly high and the implementation in software is not fast enough for the embedded systems world. In this context, the RISC-V instruction set architecture (ISA), with its open-source form and modular design, presents a fertile ground that it is perfect for the future development. RISC-V has started in 2010, under the leadership of Professor David Patterson. RISC-V ISA is completely free and open-source, and It is based on the Reduced Instruction Set Computer (RISC) principles. This thesis is focused on the integration of cryptography extensions within the RISC-V ISA, with the dual objectives of increasing the security capabilities of embedded systems and also enhancing the perfomance compared to the simple software implementations of the algorithms. The development and evaluation of specialized cryptography extensions for the RISC-V ISA take center stage, considering the crucial balance between power consumption, area, and performance. Indeed the thesis presents a comprehensive analysis of the design, implementation, and optimization of the cryptography extensions for the RISC-V processor, with a focus on the extended ISA. Utilizing tools provided by STMicroelectronics, the thesis goes through the entire process from specification, code writing, analysis and synthesis to testing, ensuring a comprehensive approach to the development of cryptography extensions. The outcome of this research goes over the academic theory, offering a contribution to the field of embedded system security. The results underline the importance of the extensions that bring an improvement on the execution speed of 5x for AES algorithm and 2x for SHA-256 and an improvement on code size of 0.5x for both the algorithms. Thanks to these results, this work is a base for future STMicroelectronics projects on RISC-V, delivering a complete development for secure, efficient, and cost-effective cryptography implementations within the RISC-V ecosystem.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Context

The evolution of computer architecture has been characterized by different historical developments. Initially, the computing landscape was dominated by the x86 architecture, developed by Intel, a complex instruction set computing (CISC) design. Another important CISC design was the 68000 Motorola processor [1]. In any case the x86 architecture dominated the market area for decades. Intel's x86 architecture, characterized by its variable-length instructions and multi-cycle operations, was well-suited to the performance demands of desktops and servers. Subsequently, a new chapter began with the ARM architecture that follows the Reduced Instruction Set Computing (RISC) architectures, characterized by a lighter design due to reduced, fixed instruction and simplified, single instruction functionality, focusing on low-power devices. Since 1980s a debate has begun between RISC and CISC, trying to decide which of the two was the best. Despite the two architectures are very different, with their main implementation focus, both the two types are present in both the implementation areas[2]. However even if ARM's success in the market was incredible, boosted by the emergence of cellphones in Europe in 1990s, its policy regarding licensing fees, presented adversities for external's customization. The model of fees and restriction on customization was advantageous for ARM but posed challenges for smaller entities and the academic research community, which

found themselves constrained by the costs and limitations imposed by proprietary architectures. During 1990 IBM introduced the IBM POWER based on RISC architecture [3]. It was in this context, some years later, that the RISC-V was born in the University of California, Berkeley in 2010. RISC-V has an ISA based on the principles of the Reduced Instruction Set Computer (RISC) and stands as a open-source design. As a open source project, RISC-V, unlike the predecessors, gives the free availability of the ISA, avoiding the payment of license fees. The architecture's simplicity, modularity and openness have not only taken the attention of academic research but caught also the interest of company leaders. The open source work is perfectly represented with the RISC-V International, a collaborative community born in 2015. A big group of stakeholders that works on the RISC-V ISA for processor design delivering a diverse suite of ratified instruction set extensions enhancing the architecture's capabilities and versatility[4][5]. Recognizing the potential of RISC-V, STMicroelectronics decided to start a project regarding the implementation of an own RISC-V-based processor aligning their main concepts to the RISC-V International topics. As in the contemporary digital epoch every embedded systems are targeted by cybersecurity attacks, and they are particularly vulnerable to such threats, STMicroelectronics recognized the critical need of robust cryptography solutions to be implemented. Starting from a software implementation of the algorithms, STMicroelectronics passed to dedicated IP integrated at SOC level due to the increasing speed needed for real-time applications and for embedded systems. In the context of the RISC-V processor, among all the extensions delivered by the RISC-V International in their specifications[6], the cryptography extensions have been delivered properly concerning the security of the processor functionality and providing security services in terms of confidentiality, data integrity, authenticity and non-repudiation. The cryptography extensions for RISC-V, implementing both Scalar & Entropy Source[7] and Vector[8] operations, are developed to optimize the performance of cryptography algorithms—fundamental to secure communication, data protection, and digital authentication. The first extension aims to improve the architecture's efficiency in executing

algorithms such as AES[9], SHA-256[10], SHA-512[10] from NIST, SM3 and SM4 from China, with support across 32-bit and 64-bit implementations. In a similar manner, also the second extension aims to improve the design regarding AES and SM2 cryptography algorithms. The integration of these extensions signifies an important advancement to reach a certain security within the processor architecture, aligning to the STMicroelectronics objectives that currently uses dedicated IP integrated at SOC level to implement this type of algorithms. Indeed my thesis is centered on the practical implementation of cryptography extensions within the STMicroelectronics processor, especially on Scalar Cryptography Extensions. Through this work, I will explore the technical challenges and solutions involved in the integration of the cryptography extensions, evaluate their impact on system performance and security, and contribute to the body of knowledge that will guide future developments in processor architecture and cryptography computing for STMicroelectronics.

1.2 Objectives

The main objectives are listed below :

1. Get practical with STMicrolectronics tools: familiarizing with software and hardware development environments, including compilers, assemblers, simulators and synthesis tools.
2. Implementation of the RISC-V Scalar Cryptography Extensions[7], more precisely:
   (a) Bitmanip instructions for cryptography: Zbkb extension to improve the performance of bitwise operations.
   (b) Crossbar permutation instructions: Zbkx extension to support complex bit-level permutations that are used in cryptography algorithms.
   (c) SHA-256 and SHA-512 instructions: Zknh extension developed for hashing algorithms that improves the processor's ability to perform secure cryptography hashing operations efficiently.

   (d) AES instructions: Zknd and Zkne extensions implementing the Advanced Encryption Standard (AES) to execute fast and secure data encryption and decryption.

3. Area and Performance evaluation through:

   (a) Synthesis Report: analyzing the impact of the extensions on the hardware implementations regarding silicon area.

   (b) Benchmarks: analyzing the speed and resource utilization during the execution of a cryptography algorithm giving an efficiency feedback.

4. Implementation optimization following Area and Performance results: improving the design to reach the maximum performance with minimum area and minimum power consumption possible.

1.3 Organization

The thesis is organized as follows:

1. Chapter 1: Introduction
2. Chapter 2: Background
3. Chapter 3: STxP5 Processor
4. Chapter 4: Tools and Workflow
5. Chapter 5: Scalar Cryptography Extension
6. Chapter 6: Other Processor Crypto Extension
7. Chapter 7: Simulation and Formal Verification
8. Chapter 8: Synthesis
9. Chapter 9: Benchmarks
10. Chapter 10: Conclusions and Future Works

Chapter 2

Background

2.1 RISC-V

This chapter gives an overview regarding RISC-V architecture, required to comprehend the context in which this thesis is placed. It begins by describing the RISC-V Instruction Set Architecture (ISA) with its base extension and all the other available extensions, highlighting its core design principles. Moreover, the instruction formats will be analyzed. Through this chapter, readers will gain the background needed to have a critical reference point for the subsequent chapters, which fall into the implementation of cryptography extensions and their integration into a RISC-V-based processor.

2.1.1 Instruction Set Architecture

The RISC-V processor has three different available versions: 32-bit (RV32), 64-bit (RV64) and 128-bit (RV128), all are load-store architectures. It is specified mainly by 2 specifications : the unprivileged specification [11] (that aims at describing the instructions - encoding and what they do) and the priviledged specficication [12] (that aims at describing the exceptions and other control feature). Differently from previous processors, RISC-V is not defined by a fully specified ISA, but it consists of a base integer ISA and optional extensions allowing the modular extensibility of any implementation. The base integer ISA

must be present in any implementation and it provides an essential set of instructions sufficient to provide a minimal target for compilers, assemblers, linkers, and operating systems[12]. So this base ISA is a simple starting point ISA around which it is possible to add extensions incrementing the complexity of the ISA. This extension mechanism is very useful for developing general-purpose solutions and avoiding the implementation of useless operations. The extensions defined by RISC-V ISA in the first version are detailed in Table 2.1. Then, with

I	Base Integer Extension
E	Reduced Base Integer Extension
M	Extension for Integer Multiplication and Division
A	Extension for Atomic Instructions
F	Extension for Single-Precision Floating-Point
D	Extension for Double-Precision Floating-Point
C	Extension for Compressed Instructions

Table 2.1: RISC-V ISA First Version

the birth of the RISC-V International, the development of the ISA has rapidly progressed providing a huge variety of extensions. All the new ratified extensions are listed in Table 2.2. Regarding privileged instruction sets, the ISA can support three different privileged levels modes [12]:

1. Machine mode (M-mode): the unique mandatory mode, used for simple embedded systems.

2. User mode (U-mode): used with Machine mode to have a secure embedded systems.

3. Supervisor mode (S-mode): added to M,U modes to have systems running Unix-like operating systems.

In addition to the three modes, there is also a fourth mode: the Debug mode (D-mode), used to support off-chip debugging and/or manufacturing tests [12].

2.1.2 Instruction Format

The RISC-V architecture uses an efficient instruction format, which is crucial for the design and implementation of the processor [11]. It is important to underline the absence of status bits as Z (zero), N (negative), C (carry) in the design of the architecture which It is unusual but It has been a decision to keep the simplicity and flexiblity of the architecture. Focusing on the RV32I, the instruction length is fixed at 32 bits and based on this, there are different formats to describe the base integer instruction set for the identification and execution of instructions. These formats are:

- R-type: used for register-register operations.
- I-type: used for immediate operations and register loads.
- S-type: used for store operations.
- SB-type: used for conditional branch instructions.
- U-type: used for instructions that load a 20-bit immediate into the upper 20 bits of a register.
- UJ-type: used for jump instructions.

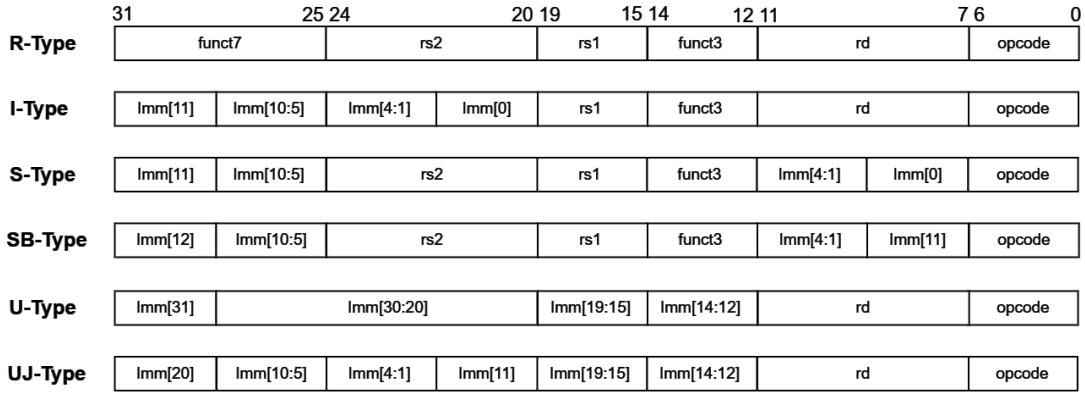

Figure 2.1: RISC-V Instruction Formats

As in figure 2.1, instruction format consists of several key points, including the opcode, funct3, and funct7, which are fundamental for identifying and decoding the instruction. Furthermore rs1, rs2, and rd (5-bit each) represent the addresses of the source 1, source 2, and destination registers, respectively. So the instruction in the processor works with the values stored in rs1 and rs2 as data inputs and writes the data output in register rd. Note that rs1, rs2 and rd are in the same position, in all the formats, to optimize the chip circuitry and increase efficiency. Furthermore, the imm field provides a binary value directly embedded within the instruction, with variable length. Therefore it provides a constant data for immediate operations for which the rs2 register is not needed and sometimes also the rs1 register is not needed as well. Regarding binary values given directly in some instruction format, the immediate can be also divided in more than one piece. These variants offer flexibility and adaptability in addressing a diverse range of computational requirements. Focusing on the registers previously called, the RV32I has 32 general-purpose registers, denoted as x0 through x31. Each register can store a 32-bit value, aligned with the 32-bit architecture. The large number of available registers can improve performance through less memory accesses. Another optimization is the hardwired register x0 to 0 through which the instructions can be simplified and also the number of needed immediate is reduced. Instead the RV32E configuration has 16 registers for PPA (Power, Performance, Area) optimizations.

2.1.3 RV32I Base Integer Instructions

Focusing on the base integer extension, precisely on the RV32I, a variety of instructions are provided[11]. The available instructions can perform arithmetic calculations, logical operations, and managing data flow within a program. Understanding these instructions is necessary, as they form the basis for more complex algorithms and extensions.

Integer Computational Instructions

Integer computational instructions cover principally arithmetic instructions. These instructions operates on 32 bits integer values stored in the integer register file. Integer instructions can have three different formats:

- R-type: R-type instructions involve register-register operations, where the operation is performed between two source registers (rs1 and rs2), and the result is stored into a destination register (rd). Operations as ADD, SUB, SLT, SLTU, AND, OR, XOR, SLL, SRL and SRA are part of this type. For example, the ADD instruction adds the values in registers rs1 and rs2 and stores the result in register rd. Similarly, the SUB instruction subtracts rs2 to rs1. SLL,SRL,SRA perform logical left, logical right, and arithmetic right shifts to the value of rs1 by an amount specified by the lower 5 bits of the value in rs2. Moreover XOR,OR,AND apply bitwise operations between rs1 and rs2. For comparison operations, SLT(signed) and SLTU(unsigned) compares the content of rs1 with rs2 and put 1 as result in rd if rs1 is less than rs2, otherwise a 0 is stored. In these specific scenarios, the absence of the status bits is significant, as the status bit is meant for that purpose.

- I-type: I-type instructions involve register-immediate operations, where an immediate value (12-bit value) is sign-extended to be used correctly with the other 32/bit value and then used as one of the operands in the operation. Instructions as ADDI, SUBI, SLTI, SLTIU, ANDI, ORI, XORI, SLLI, SRLI, SRAI are included in this category. The operations performed by these instructions are the same of the previous with the immediate value instead of the rs2 register.

- U-type: U-type instructions as LUI and AUIPC take part of this type. LUI performs a load of a 20-bit immediate value into the upper 20 bits of the destination register while the lower 12 bits are filled by zeros. Instead AUIPC takes the 20-bit immediate value as upper bits and filling the other 12 bits with zeros. Then It uses the

value as an offset to be added to the program counter (PC) and loads the result of the addition in rd.

Control Transfer Instructions

Control transfer function, to control the flow of the execution, is divided in two parts: unconditional jumps and conditional branches. Unconditional jumps have the JAL instruction which follows the UJ-type. This instruction performs a jump using a signed offset, taken from the instruction, that is added to the program counter(PC) to form the jump target address. In the meanwhile the instruction can store the theoretical next address(PC+4) that is the return address in a specific register rd (x1). There is also the J pseudo-instruction that is a JAL without storing the return address, and so it in this case the rd is x0. In addition, the JALR instruction, that follows the I-type format, performs the jump calculating the target address by adding the 12-bit signed immediate with the value of the register rs1 and the putting a zero in the least significant bit of the result. As in JAL, the return address is stored in a specific rd. Conditional branches use the SB-type format, and compare the contents of the two register and if the condition is positive, PC will be updated with the target address calculated by adding the 12-bit signed immediate to the program counter. The BEQ and BNE instructions update the PC if the registers are equal or not equal. The BLT and BLTU instructions update the PC if the first register is less than the second, using signed or unsigned comparison. Similarly, BGE and BGEU update the PC if the first register is greater than or equal to the second, using signed or unsigned comparison.

Load and Store Instructions

Load and store instructions can work on the memory and the register file. These instructions are used to transfer values between registers and memory and viceversa. The load instruction, that follows the I-type format, declare LW[U](32-bit words), LH[U](16-bit half-words), LB[U](8-bit bytes). These instructions perform a load from memory to the rd register, in which the memory address is computed with

the addition of the value of rs1. With a value of 16-bit or 8-bit, an extension of zeros is done with -U declared, otherwise a sign-extension is done. The store instruction, which follows the S-type format, includes SW(32-bit words), SH(16-bit half-words), SB(8-bit bytes) perform a store from a register to the memory with address the addition between rs1 and the immediate. Instead the value stored is taken from the register file with the address rs2.

2.2 Cryptography

The advent of computers, networks and the increased dependency on digitized information in our society makes information more vulnerable from cyber-attacks. For this reason, it is important to secure information systems by protecting data and resources from malicious acts through cryptography algorithms. Cryptography offers a robust solution for IT security by providing security services in terms of confidentiality, data integrity, authenticity and non-repudiation [13]:

- Confidentiality: Information cannot be accessed by unauthorized parties. This is accomplished through public key and private-key encryption.
- Data Integrity: Transmitted data within a given communication cannot be altered during storage or transmission. This is done through Hash Functions.
- Authenticity: Ensures that, within a given communication, the source of information and the information itself are genuine though digital signatures.
- Non-repudiation: Ensures that neither the sender nor the receiver of a message can deny transmission. This is achieved via digital signatures and third party notary services.

In this context cryptography algorithms such as AES (Advanced Encryption Standard) and SHA-256/512 have an important role in ensuring the characteristics specified before.

2.2.1 Importance of Cryptography Extensions

Implementing cryptography algorithms in software is relatively easy, but such algorithms are typically too slow for real-time applications, such as storage device and embedded systems, due to increasing data rates and complexity of security protocols. For this reason, it becomes necessary to implement cryptography algorithms in hardware. Hardware-based cryptography significantly accelerates the execution of cryptography algorithms by providing dedicated instructions into the processor architecture, thus enhancing the overall system performance while maintaining high levels of security [14] . In this context, one notable example is the integration of the RISC-V cryptography extensions, specifically the scalar cryptography extensions, which have been designed to support efficient cryptography operations on 32-bit and 64-bit RISC-V processors. These extensions include specialized instructions for AES encryption and decryption, as well as SHA-256 and SHA-512 hashing, which are commonly used in securing communications and protecting sensitive data. There are advantages thanks to hardware-based cryptography implementations [15]:

- Performance: The instructions can exploit all the capabilities of the processor, significantly speeding up the cryptography operations.
- Attack surface: The hardware implementation can hide implementation details and reduce the attack surface.
- Memory: The algorithms in hardware can reduce the code size and also the data-memory.

All these advantages of the hardware implementation make it a proper solution to integrate these functions as ISA extensions and so implement them in hardware. In the case of AES and SHA-256, comparing the software implementation with the hardware implementation, there is an improvement of 5x and 2x on speed and 0.5x on program memory, respectively. Since the instructions are implemented directly in the processor, there is a disadvantage in term of area, 1.1 kGates for AES and 0.7 kGates for SHA-256/512 [16] [17] [18]. These are the expected

results, all the instructions, that are part of the Scalar Cryptography Extensions, and all the results will be explained in the next chapters.

Zifencei	Instruction-Fetch Fence
Zicsr	Control and Status Register Instructions
Zicntr	Counters
Zihintntl	Non-Temporal Locality Hints
Zihintpause	Pause Hint
Zimop	May-Be-Operations
Zicond	Integer Conditional Operations
Zmmul	Multiply only
Zawrs	Wait-on-Reservation-Set Instructions
Zacas	Atomic Compare-and-Swap Instructions
RVWMO	Memory Consistency Model
Ztso	Total Store Ordering
CMO	Base Cache Management Operation ISA
Q	Quad-Precision Floating-Point
Zfh	Half-Precision Floating-Point
Zfhmin	Minimal Half-Precision Floating-Point
Zfa	Additional Floating-Point Instructions
Zfinx-Zdinx-Zhinx-Zhinxmin	Floating-Point in Integer Registers
Zc*	Code Size Reduction
B	Bit Manipulation
V	Standard Extension for Vector Operations
Zbkb	Bitmanip instructions for Cryptography
Zbkc	Carry-less multiply instructions
Zbkx	Crossbar permutation instructions
Zk	Standard scalar cryptography extension
Zks	ShangMi Algorithm Suite
Zvbb	Vector Basic Bit-manipulation
Zvbc	Vector Carryless Multiplication
Zvkg	Vector GCM/GMAC
Zvkned	NIST Suite: Vector AES Block Cipher
Zvknhb	NIST Suite: Vector SHA-2 Secure Hash
Zvksed	ShangMi Suite: SM4 Block Cipher
Zvksh	ShangMi Suite: SM3 Secure Hash
Zvkt	Vector Data-Independent Execution Latency

Table 2.2: RISC-V International Ratified Extensions

Chapter 3

STxP5 Processor

This chapter describes the main characteristics of the STxP5 processor, recalling some information of the Chapter 2 and giving some other information. STxP5 architecture is one implementation of the standard RISC-V free open architecture. STxP5 architecture is shown in Figure 3.1. STxP5 processor is made of 5 main functional blocks :

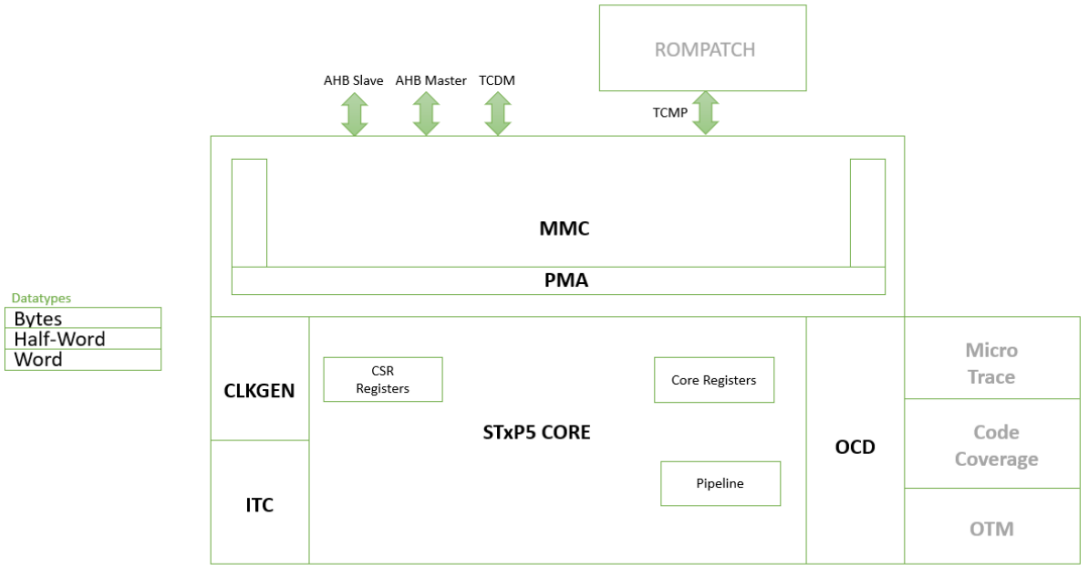

Figure 3.1: STxP5 processor

- The STxP5 core. It is in charge of fetching STxP5 instructions and

executing them.

- The memory controller (MMC). It is in charge of the arbitration between the instruction fetch and data load/store requests which are converted into the AHB-lite protocol.

- The interrupt controller (ITC). It provides the possibility of handling interrupts.

- The on chip debug (OCD). It provides the necessary resources and control for the software debug on the STxP5 HW target.

- The physical memory attributes (PMA). It holds attributes that are checked by PMA_checker and that can raise exceptions when violated.

In Figure 3.1, it is also shown the possible data types for the STxP5. The STxP5 provides an efficient support for 8-bit, 16-bit, 32-bit signed and unsigned data types. While all computational operations are performed on 32-bit, the other smaller data types are promoted to a 32-bit while they are loaded from memory. The 32-bit results are down-cast when the result is stored into memory. In Figure 3.2 is shown how the data types are ordered in memory.

<table>
<tr><td>Address+n</td><td>...</td><td>Address+4</td><td>Address+3</td><td>Address+2</td><td>Address+1</td><td>Address</td></tr>
<tr><td></td><td></td><td></td><td>Byte3</td><td>Byte2</td><td>Byte1</td><td>Byte0</td></tr>
<tr><td></td><td></td><td></td><td colspan="2">HALFWORD1</td><td colspan="2">HALFWORD0</td></tr>
<tr><td></td><td></td><td></td><td colspan="4">WORD0</td></tr>
</table>

Figure 3.2: Bytes ordering in memory

3.1 STxP5 core

As the objectives of the thesis is regarding the implementation of different instructions, it could be important to focus on the STxP5 core.

3.1.1 Core Registers

STxP5 general purpose register file X can be configured to implement either 16 or 32 32-bit registers giving the possibility to implement either RV32I or RV32E base ISA. The register file is a RISC-V standard and It is described in Table 3.1. As said previously, also here, the register X0 is hardwired to 0, meaning that it can not be overwritten. The link register X1 holds the return address of a function call. The stack pointer X2 must be set to the base address of the stack by software and it is growing downward. Temporary registers are registers which do not hold their value across function calls, they must be saved by the caller before a call if they are later used. Instead the saved registers are saved by the caller if they are used within the function, their value is thus held across function calls. X10 and X11 can be used for dual purposes: function arguments and/or to return function results. Registers X12 to X17 can be used only for function arguments. The register file configuration interacts with the Application Binary Interface (ABI). The ABI is a convention that tells how software shall use the registers during function calls, return address handling and parameter passing mechanism.

3.1.2 Control and Status Registers

STxP5 supports two privileged modes, so only a subset of the privileged modes of RISC-V . It implements the Machine Mode (M-mode) that is the highest privileged mode. This mode has complete authority over CPU scheduling and configuration. Moreover, execution state of all exceptions and interrupts is M-mode. The second mode is the User Mode (U-mode) that is used for conventional usage. So the Supervisor Mode (S-mode) is not supported. Table 3.2 lists the main Control and Status registers (CSRs) implemented. The CSRs address space is sets aside a 12-bit encoding space (csr[11:0]) for up to 4,096 CSRs. Conventionally, the upper 4 bits of the address (csr[11:8]) are used to described the read and write accessibility according to the different privilege levels. More precisely the top two bits (csr[11:10]) indicate whether the register is read/write with 00, 01, or 10 or read-only with

11. Then the other two bits (csr[9:8]) encode the lowest privilege level that can access the CSR[12].

Therefore for each register, the possible access to the register is given:

- MRW: Machine Read Write.
  - Read and Write accesses are supported in Machine Mode.
  - Any access to the register in User mode will generate an illegal instruction exception.
- MRO: Machine Read Only
  - Read access is supported in Machine Mode.
  - Any access to the register in User mode will generate an illegal instruction exception.
  - Write access to the register will generate an illegal instruction exception
- RW: Read Write
  - Read and Write accesses are supported in User Mode and Machine Mode.
- RO: Read Only
  - Read access is supported in User Mode and Machine Mode
  - Write access to the register will generate an illegal instruction exception

3.1.3 Pipeline

The STxP5 is a 4-stage pipeline implementation of RISC-V and so an instruction can be executed in a maximum 4 pipelined cycles. The 4-stage instruction pipeline is as follows:

- **Stage1: Instruction Fetch (IF)**

  Instruction is predecoded. Predecoding is performed on either the new 32-bit chunk of data read from memory or on the next 32-bit chunk read from a 80-bit instruction prefetch buffer. If the 32-bit correspond to a 32-bit instruction it is registered in the 32-bit instruction register for the next ID pipeline stage. Otherwise the 16-bit opcode is zero extended before being registered into the instruction register.

- **Stage2: Decode, Operand Fetch (DOF)**:

  The instruction is decoded. Instruction operands are read from the register file. In case of a load or store instruction, the effective address (EA) of the data to read or write is computed and registered. Unconditional branches are providing the relative branch offset to the control unit in this cycle too

- **Stage3: Execute (EX)**:

  The Result of instructions belonging to ALU, SHIFT, MUL instruction classes is computed and registered. Memory read or write is performed. If branch condition is true, branch address is sent to the control unit.

- **Stage4: Write Back (WB)**:

  The Result of instructions belonging to ALU, SHIFT, MUL, DIV instruction classes is written back into the register file. The Result of Pre and post modifying addressing modes is written back into the register file. Memory read is completed and the data load from memory is written back into the register file.

Control Unit

The Control Unit is in charge of managing the datapath throughout its stages. Given an instruction from instruction register(IR), it generates the corresponding control word that manages the various registers, MUXes and other control signals in the datapath. Moreover, the CU

interacts with the Hazard Unit (HU), which monitors the pipeline for potential hazards that could break the flow of instructions. The HU provides signals to the CU indicating when it is necessary to stall the pipeline or insert bubbles (NOP operation) instructions to allow time for data to become available or for previous instructions to complete.

Hazard

The Hazard Unit takes care of detecting data control and structural hazards in the pipeline and successively dispatching the correct signals to the CU to indicate where and for how many stages to stall for. If no hazardous condition is found, the pipeline operates uninterrupted, maintaining the flow of instruction execution. Three main cases are covered by the HDU:

- Data hazards : pipeline stalls when an instruction depends on the result of a previous instruction that has not yet finished, ensuring correct program execution without data corruption.
- Structural hazards: pipeline has to stall when an instruction needs an hw resource that is busy by the previous one.
- Control hazards: pipeline has to be flushed because with a jump or a taken branch, the normal flow of execution is changed.

Forwarding

To mitigate the impact of hazards, for data hazards, techniques such as forwarding (or bypassing) can be used to pass the result of a computation directly to a subsequent instruction without writing it to and reading it from the register file. This is implemented in order to reduce the need for stalls and keeps the pipeline moving. The Forwarding Unit detects favourable conditions for source and destination registers between stages of the pipeline and if they match it forwards operands where they are needed, skipping the write-back stage. The forwarding unit helps improving the pipeline performances, reducing the number of stalls due to hazards in different situations. The situations in which

the forwarding can be exploited are when an instruction has already produced the result without writing that in the register file, and a following instruction needs the data in the execute stage. The majority of forwarding paths links write-back stage with the execute stage.

A simplified datapath of the pipeline processor is shown in Figure 3.3

32-bit Register File X	**Alias name**
X0: Zero (hardwired 0)	zero
X1: Link register	ra
X2: Stack pointer	sp
X3: Global pointer	gp
X4: Thread pointer	tp
X5: Temp	t0
X6: Temp	t1
X7: Temp	t2
X8: Saved register	s0
X9: Saved register	s1
X10: Fct arg/Ret val	a0
X11: Fct arg/Ret val	a1
X12: Fct argument	a2
X13: Fct argument	a3
X14: Fct argument	a4
X15: Fct argument	a5
X16: Fct argument	a6
X17: Fct argument	a7
X18: Saved register	s2
X19: Saved register	s3
X20: Saved register	s4
X21: Saved register	s5
X22: Saved register	s6
X23: Saved register	s7
X24: Saved register	s8
X25: Saved register	s9
X26: Saved register	s10
X27: Saved register	s11
X28: Temp	t3
X29: Temp	t4
X30: Temp	t5
X31: Temp	t6

Table 3.1: STxP5 Register File X

CSR Address				Hex	Acc.	Name
[11:10]	[9:8]	[7:6]	[5:0]			
00	11	00	000000	0x300	MRW	Machine Status (mstatus)
00	11	00	000001	0x301	MRW	Machine ISA and extension (misa)
00	11	00	000100	0x304	MRW	Machine interrupt-enable (mie)
00	11	00	000110	0x306	MRW	Machine counter enable (mcounteren)
00	11	00	000101	0x305	MRW	Machine trap vector base-address (mtvec)
00	11	01	000001	0x341	MRW	Machine exception program counter (mepc)
00	11	01	000010	0x342	MRW	Machine trap cause (mcause)
00	11	01	000100	0x344	MRW	Machine interrupt pending (mip)
11	11	00	010010	0xF12	MRO	Machine architecture ID (marchid)

Table 3.2: Control and Status Registers(CSRs) list

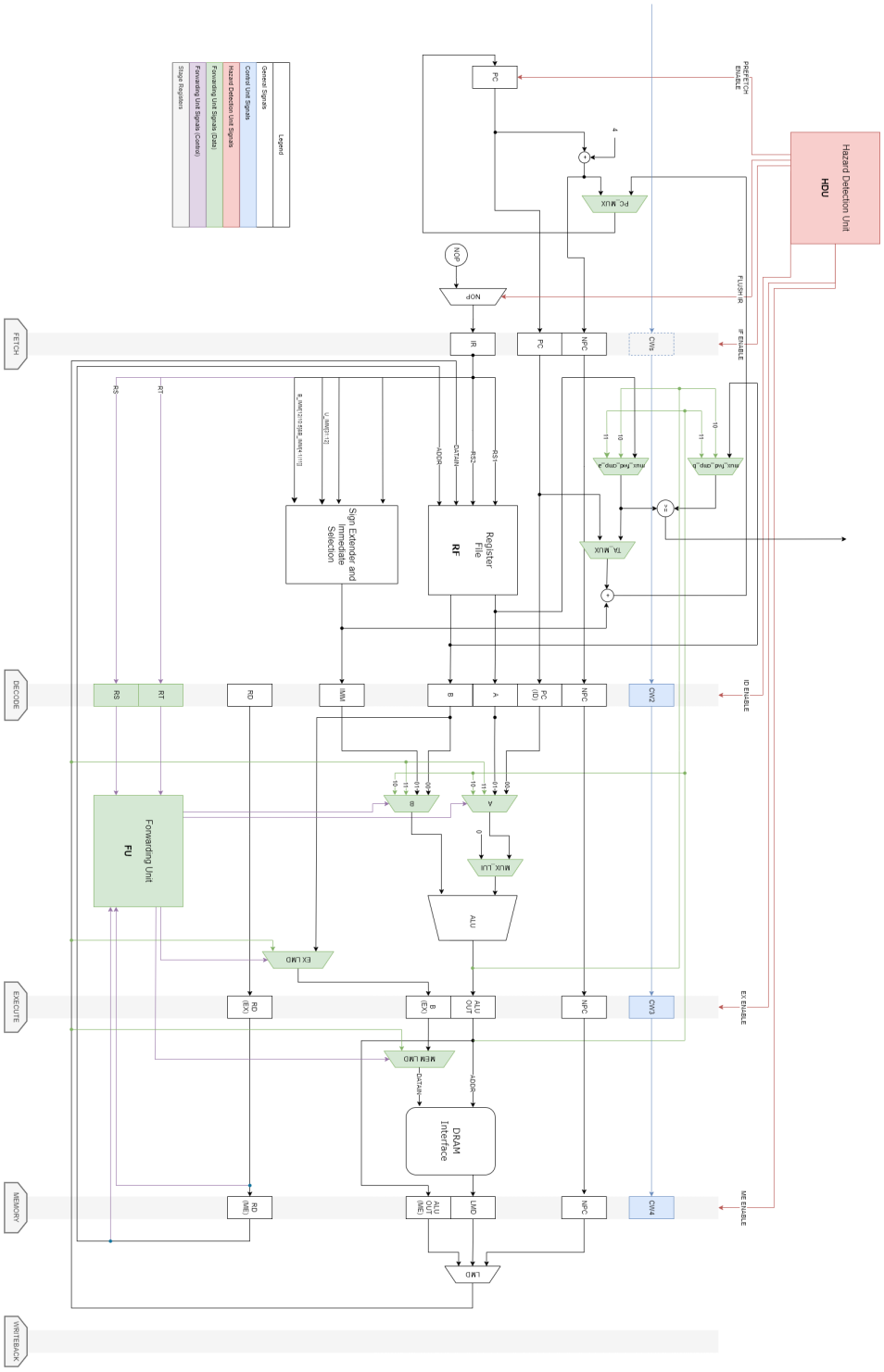

Figure 3.3: Simplified STxP5 Datapath

Chapter 4

Tools and Workflow

4.1 ASIP Tool

In the world of data processing, a crucial tradeoff exists between flexibility and efficiency. The Figure 4.1 shows the different possible solutions [19]. In the leftmost position, there are general-purpose microprocessors that they are the most flexible solution but with the drawback of low performance and low power efficiency. On the other hand, in the rightmost solution, hardwired datapaths are the most efficient solution but with a singular application possible without the possibility to be adapted to different applications. In between these two, there is the Application Specific Integrated Processor (ASIP) that offers a balance between the two parameters. Synopsys' ASIP Designer is a software tool for the design and programming of Application Specific Integrated Processors (ASIPs) [20]. This suite has been developed to support any user-modeled processor architecture described in nML (notation Machine Language). The nML language captures both the instruction-set architecture (ISA) at a high-level and the microarchitecture of a processor at a Register Transfer Level (RTL). It can be viewed as a specialized hardware description language, but with additional functions specifically designed to describe processor features for particular application domain. This language facilitates the modeling and the optimization of a processor's architecture. The nML model is the fundamental input for ASIP Designer's retargetable software development kit

(SDK). This SDK is a powerful software, It includes an optimizing C and C++ compiler that enforces the capabilities of the ISA,as defined in nML, to its fullest potential, with also a linker and an assembler. The compiler's ability to exploit the full potential of the ISA is thanks to the efficacy of ASIP Designer's Compiler-in-the-Loop design methodology. Starting from a code written in C or C++, the behavior of the processor model can be simulated. This simulation is executed through a cycle-accurate instruction-set simulator (ISS) within the SDK, plus a graphical debugger and profiler. The advantage of the of the cycle accurate simulation, with the retargetable SDK, is the possibility of tuning the ISA, changing the nML code, and validate it in a next iteration cycle. Moreover, ASIP tool includes a RTL generator that It is capable of translating the nML model into a Register Transfer Level (RTL) model either Verilog or VHDL languages. The generated code is optimized to achieve good timing performance and power dissipation. The immediate generation of an RTL model from the nML description is a cornestone of the Synthesis-in-the-Loop methodology. Indeed, to perform a deeper analysis of the ASIP's performance, the RTL can be synthesized. Thanks to the synthesis, designers can study and evaluate the power, performance and area (PPA) of the design. As last step, there is the verification process. The model is tested with user-defined algorithm written in C or C++ on three different levels: firstly the model is tested on the host compiler, as a reference, after It is tested with the ISS of the model and, as last level, It is tested with the RTL Simulator. All the three results will be compared to underline possible mismatches or full correctness. The full ASIP designer tool flow is given in Figure 4.2.

4.1.1 nML and PDG

In the design process a processor's architecture is defined through the high-level nML language that can described both the instruction-set architecture and the microarchitecture. The main characteristics are listed below [21] :

- Primitive data types and primitive operations: They are the core of

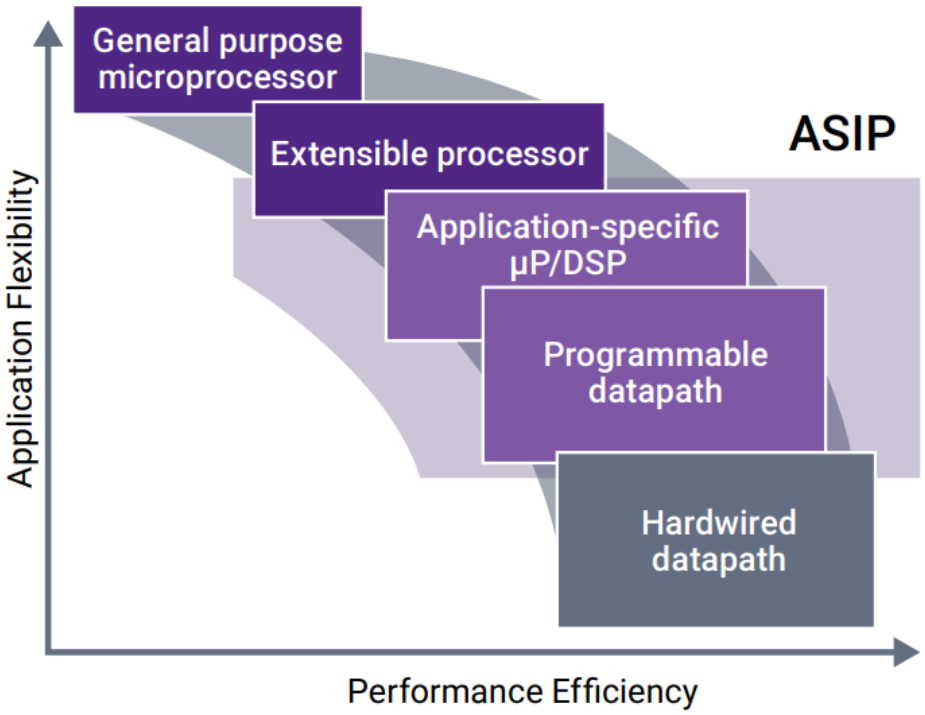

Figure 4.1: Flexibility and Performance Tradeoff

the model, they can be standard or user-defined. The primitive data types are modeled using C++ classes. The primitive operations are modeled using C++ functions and operators.

- Data storages: All of them must be declared globally with its data type and address type.

  - Static storages: They can store each of its values during several machine cycles, until They are explicitly overwritten.

    * Memories: They are storage elements that usually have the possibility to store a large number of values at the same time. Access operations can take relatively long compared to normal machine operations. In load-store architectures, memory may only be accessed by special load and store instructions, not by arithmetic instructions.

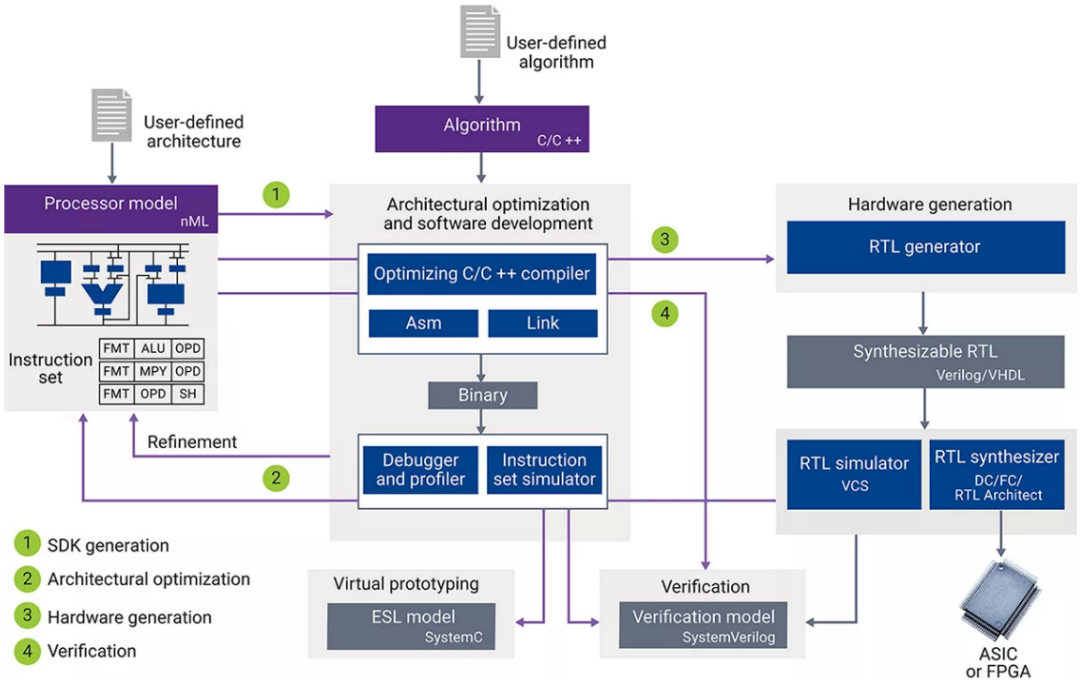

Figure 4.2: ASIP Designer Flow

    * Registers: They are accessible in much less time than a machine cycle. In load/store architectures, arithmetic operations always take their operands from registers and put their results in registers.

  – Transitory storages: They represent buses or wires. They pass a value from input to output without delay.

    * Processor ports: The input and output ports of the processor modeled by transitories. An input transitory can only be read and may not be written , while an output transitory can only be written and may not be read.
    * Pipeline registers: Special transitories with delay of one cycle. These transitories are used to describe multi-stage actions.

- Functional units: They can optionally be declared. Functional

units are just used to group operations that are physically executed on the same hardware unit.

- Immediate and Hardwired Constants: Every immediate constant must be declared instead an hardwired constant must not be declared.
- Properties: They are used to specify the special purpose of some storage elements.
- Instruction-set: It describes the instruction set and the execution behavior of the processor. It is detailed through several rules with specific grammar and attributes. Instructions are defined through:
  - The 'action' attribute, which details the concurrent register-transfer operations for all instructions within the class, organized into stages of the processor's pipeline.
  - The 'syntax' attribute, which outlines the assembly language representation of the instructions.
  - The 'image' attribute, which specifies the binary encoding of the instructions.

While nML describes the structural framework of the processor, the behavior of primitive functions is defined in the PDG (Primitives Definition and Generation) language [22]. PDG is a hybrid language combining C and HDL elements like Verilog, used for defining functional primitives and control units. It utilizes the primitive data types from nML and enhances them with additional features. The PDG language is crucial as it avoids inconsistencies and duplication of effort by enabling the generation of C++, VHDL, and Verilog implementations from a single definition. It also describes other processor components, such as the Processor Control Unit and I/O interfaces. In summary, the nML and PDG languages within ASIP Designer provide a comprehensive framework for defining a processor's architecture, from its data types and operations to its functional units and instruction set, ensuring consistency and efficiency in the design and simulation of ASIPs. Appendix A gives an example of PDG-nML code.

4.2 Workflow

The STxP5 team carries out its IP development in a Linux environment. The projects managed with the Git version control tool and is divided mainly into three repositories: architecture, design and verification. The architecture git repository contains the description of the different processor blocks, scripts and compilation tools etc. The design repository contains all scripts, constraint files and tools necessary for synthesis and performance measurements (PPA) of the processor. Finally, the verification git as its name suggests, contains the tools and tests necessary for design verification. These different deposits constitute a fairly vast environment and complex files. The STxP5 project organization is showed in Figure 4.3.

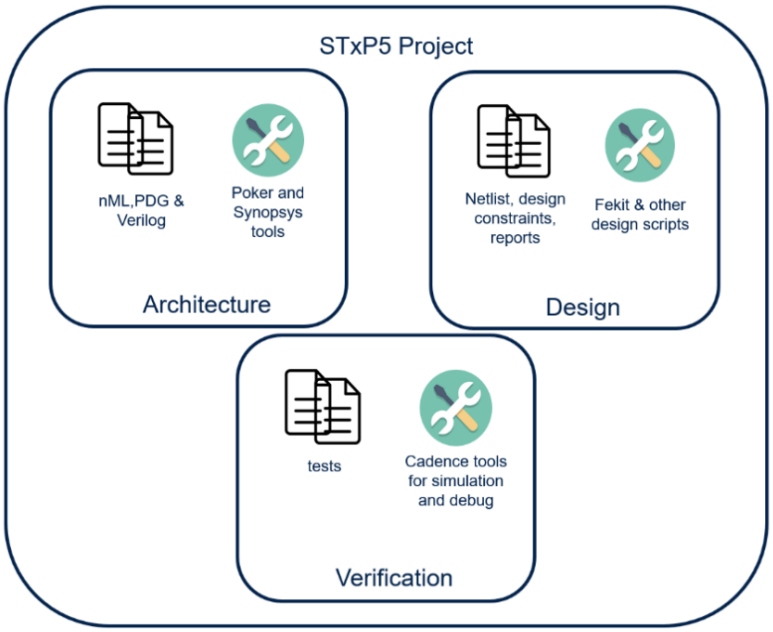

Figure 4.3: STxP5 Organization Project

The implementation of the designs is carried out from nML and PDG. The files are compiled with Chess (C Compiler), another ASIP tool. It is possible to do this directly at from command lines or through internal software called Poker, tool developed by the team. Poker gives

the possibility, under a single graphical interface, to configure and build Processor model, prepare simulation and to run a regression. It is even possible to consult ASIP documentation via the interface. Poker, by throwing ChessDE, allows to compile the nML/PDG then generates the Verilog. This Verilog output is then used in a standard CAD flow to run simulation (Xcelium rtl simulator), to be synthesized using a targeted standard cell library (40 nm), to run gate simulation to extract power estimation. These tasks can be carried out via an online tool called Jenkins which centralizes, automates and improves the presentation of the results following the continuous integration strategy. To ensure the quality of the design, verification work is generally carried out by an engineer from the ST verification team in collaboration with the designer. The test plan is to write dynamic and formal tests that validate the implementation. Dynamic verification ensures that features have been implemented correctly, while formal verification ensures that the implementation agrees with the behaviors described by architecture specifications or protocols. These two methods are complementary to demonstrate the quality of the design. The STxP5 flow is showed in Figure 4.4.

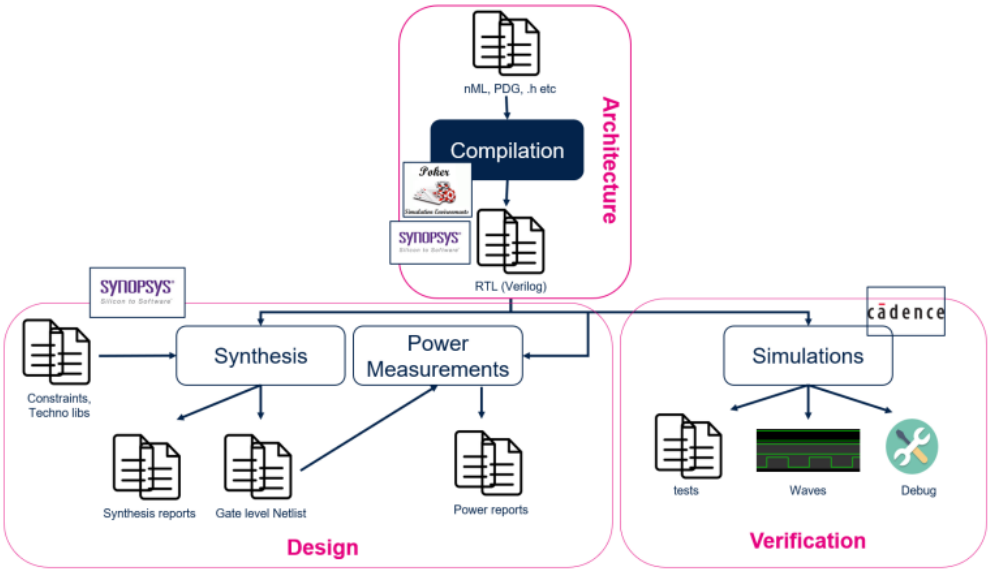

Figure 4.4: STxP5 Flow of Design

Chapter 5

Scalar Cryptography Extension

This section explains the hardware architecture of the Scalar Cryptography Extension. This extension is divided in more than one group that They are showed in Figure 5.1. My thesis is focused on the implementation of the Zkn group except the Zbkc that It is not a priority for STxP5 in this moment. The Zkn extension is built for Nist algorithhm.

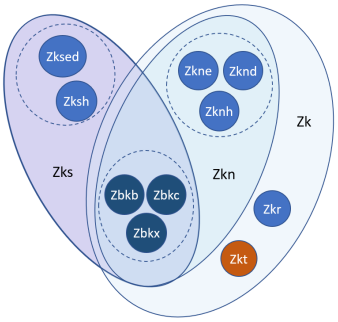

Figure 5.1: Scalar Cryptography Groups [23]

5.1 Specifications

All instructions described herein use the general-purpose X registers, and obey the 2-read-1-write register access constraint. These instructions

are designed to be lightweight and suitable for 32-bit base architectures [7].

5.1.1 Zbkb - Cryptography Bitmanip instructions

These instructions are a subset of the Bitmanipulation Extension Zbb which are particularly useful for Cryptography. There are 5 instructions that are not present in the Zbb extension, so They have been implemented and added to the others.

Reverse instruction

- **brev8 rd, rs** : Reverse bits in bytes, 5.2.

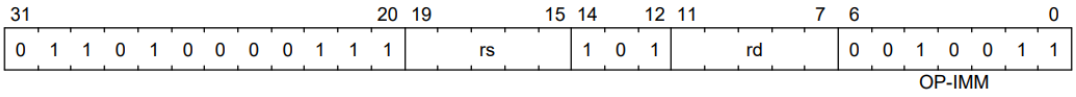

Figure 5.2: Brev8 Encoding [7]

Packing instructions

- **pack rd, rs1, rs2** : Pack low halves of registers, 5.3.

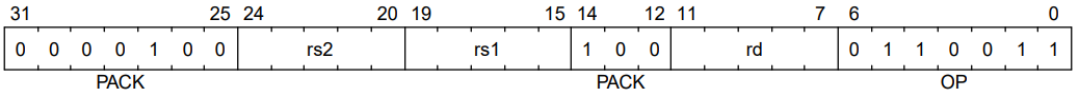

Figure 5.3: Pack Encoding [7]

- **packh rd, rs1, rs2** : Pack low bytes of registers, 5.4.

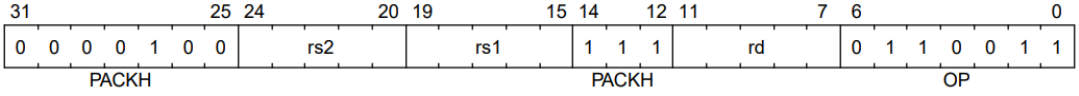

Figure 5.4: Packh Encoding [7]

Generalized Shuffle instructions

- **zip rd, rs** : Zip the registers, 5.5.

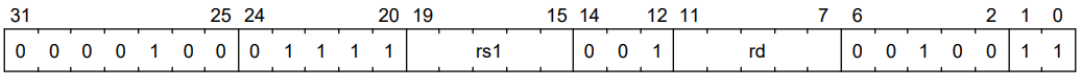

Figure 5.5: Zip Encoding [7]

- **unzip rd, rs** : Unzip the registers, 5.6.

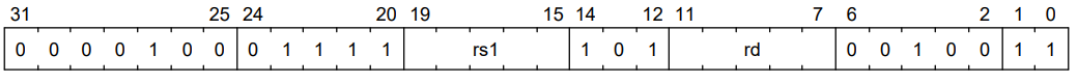

Figure 5.6: Unzip Encoding [7]

5.1.2 Zbkx - Crossbar Permutation instructions

These instructions are useful for implementing SBoxes (Appendix B) in constant time, and potentially with DPA protections.

- **xperm8 rd, rs1, rs2** : Bytes Crossbar Permutation, 5.7.

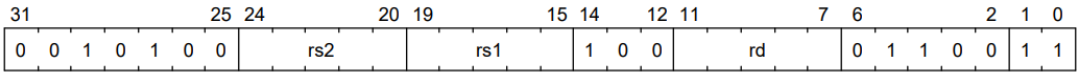

Figure 5.7: Xperm8 Encoding [7]

- **xperm4 rd, rs1, rs2** : Nibbles Crossbar Permutation, 5.8.

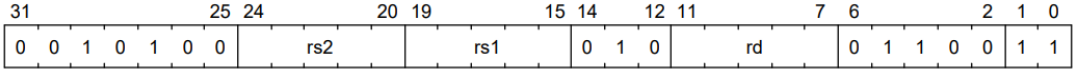

Figure 5.8: Xperm4 Encoding [7]

5.1.3 Zknh - NIST Suite: Hash Function

These instructions are implemented for accelerating SHA-2 family of cryptography hash functions. It covers the SHA2-256 functions and the SHA2-512 functions. Appendix C gives more details on SHA-2 hash family. Two logical functions (not specified) are shared between SHA-256 and SHA-512 [10].

$$\mathrm{Ch}(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z) \tag{5.1}$$

$$\mathrm{Maj}(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \tag{5.2}$$

SHA2-256 instructions

The SHA2-256 are hash functions that receive an input message with length `n` ($0 < \mathtt{n} < 2^{64}$) and generates a 256-bit output message digest. The 32-bit hash logical functions are [10]:

$$\Sigma_0 = \mathrm{ROTR}(x, 2) \oplus \mathrm{ROTR}(x, 13) \oplus \mathrm{ROTR}(x, 22) \tag{5.3}$$

$$\Sigma_1 = \mathrm{ROTR}(x, 6) \oplus \mathrm{ROTR}(x, 11) \oplus \mathrm{ROTR}(x, 25) \tag{5.4}$$

$$\sigma_0 = \mathrm{ROTR}(x, 7) \oplus \mathrm{ROTR}(x, 18) \oplus \mathrm{SHR}(x, 3) \tag{5.5}$$

$$\sigma_1 = \mathrm{ROTR}(x, 17) \oplus \mathrm{ROTR}(x, 19) \oplus \mathrm{SHR}(x, 10) \tag{5.6}$$

- **sha256sig0 rd, rs1** : SHA2-256 Sigma0 instruction, 5.9.

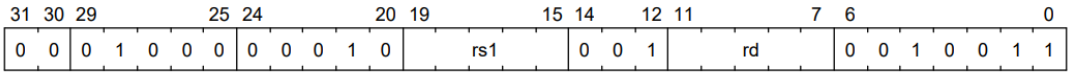

Figure 5.9: Sha256sig0 Encoding [7]

- **sha256sig1 rd, rs1** : SHA2-256 Sigma1 instruction, 5.10.

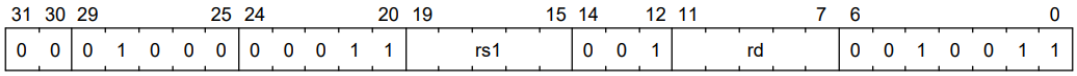

Figure 5.10: Sha256sig1 Encoding [7]

- **sha256sum0 rd, rs1** : SHA2-256 Sum0 instruction, 5.11.

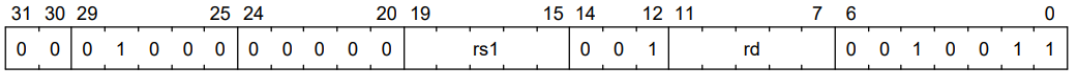

Figure 5.11: Sha256sum0 Encoding [7]

- **sha256sum1 rd, rs1** : SHA2-256 Sum1 instruction, 5.12.

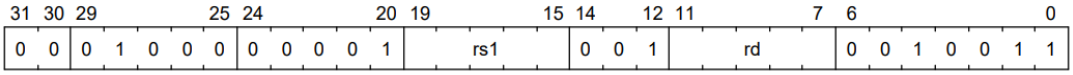

Figure 5.12: Sha256sum1 Encoding [7]

SHA2-512 instructions

The SHA2-512 are hash functions that receive an input message with length `n` ($0 < \texttt{n} < 2^{128}$) and generates a 512-bit output message digest. The 64-bit hash logical functions are [10]:

$$\Sigma_0 = \text{ROTR}(x, 28) \oplus \text{ROTR}(x, 34) \oplus \text{ROTR}(x, 39) \tag{5.7}$$

$$\Sigma_1 = \text{ROTR}(x, 14) \oplus \text{ROTR}(x, 18) \oplus \text{ROTR}(x, 41) \tag{5.8}$$

$$\sigma_0 = \text{ROTR}(x, 1) \oplus \text{ROTR}(x, 8) \oplus \text{SHR}(x, 7) \tag{5.9}$$

$$\sigma_1 = \text{ROTR}(x, 19) \oplus \text{ROTR}(x, 61) \oplus \text{SHR}(x, 6) \tag{5.10}$$

Due to the 64-bit nature of the operations and the 32-bit architecture of the processor, each function is divided into two parts, with each part processing 32-bit inputs. In the case of Sigma0 function, the instructions sha512sig0l and sha512sig0h are performed one after the other. In the same way also the Sigma1 function is implemented with sha512sig1l and sig512sig1h. The sequences are showed below:

Sigma0

```
sha512sig0l t0, a0, a1
sha512sig0h t1, a1, a0
```

Sigma1

```
sha512sig1l t0, a0, a1
sha512sig1h t1, a1, a0
```

Similarly, the Sum0 function is performed by two iterations of the sha512sum0r operation, while the Sum1 function is performed by two iterations of the sha512sum1r operation. The iterations are showed below:

Sum0

```
sha512sum0r t0, a0, a1
sha512sum0r t1, a1, a0
```

Sum1

```
sha512sum1r t0, a0, a1
sha512sum1r t1, a1, a0
```

- **sha512sig0h rd, rs1, rs2** : SHA2-512 Sigma0 high instruction, 5.13.

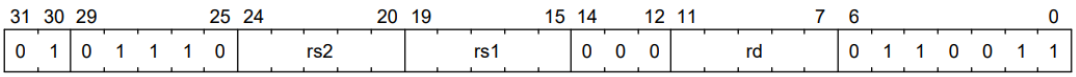

Figure 5.13: Sha512sig0h Encoding [7]

- **sha512sig0l rd, rs1, rs2** : SHA2-512 Sigma0 low instruction, 5.14.

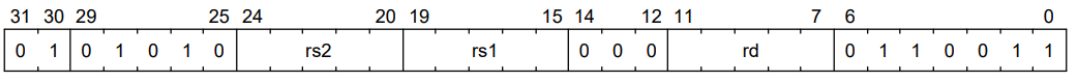

Figure 5.14: Sha512sig0l Encoding [7]

- **sha512sig1h rd, rs1, rs2** : SHA2-512 Sigma1 high instruction, 5.15.

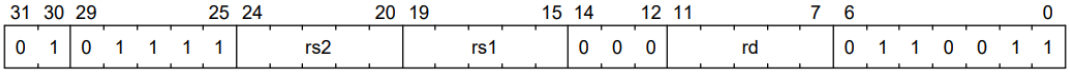

Figure 5.15: Sha512sig1h Encoding [7]

- **sha512sig1l rd, rs1, rs2** : SHA2-512 Sigma1 low instruction, 5.16.

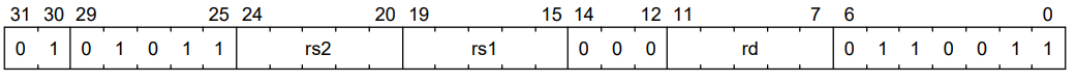

Figure 5.16: Sha512sig1l Encoding [7]

- **sha512sum0r rd, rs1, rs2** : SHA2-512 Sum0 instruction, 5.17.

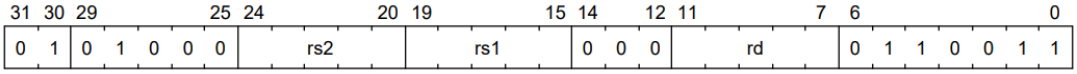

Figure 5.17: Sha512sum0r Encoding [7]

- **sha512sum1r rd, rs1, rs2** : SHA2-512 Sum1 instruction, 5.18.

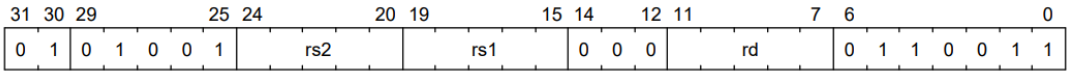

Figure 5.18: Sha512sum1r Encoding [7]

5.1.4 Zkne - NIST Suite: AES Encryption

The AES algorithmn works on 128-bit block size and three key sizes: 128, 192 and 256 bits. Depending on the key size, the number of rounds are defined: 10 rounds for AES-128, 12 rounds for AES-192,

and 14 rounds for AES-256. Each round, with the exception of the first, consists of layers, each layer manipulates all 128 bits of the input 5.19 shows the flow of the encryption algorithm.

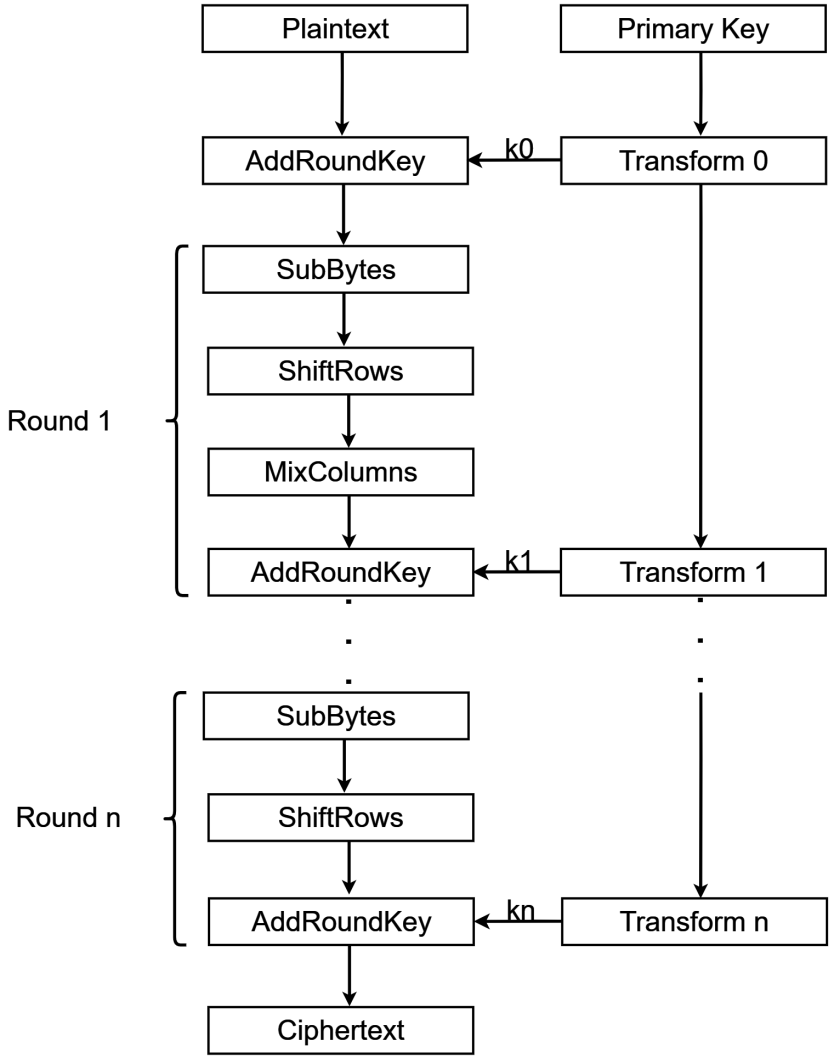

Figure 5.19: AES Encryption [24]

The layers are described below:

- **SubBytes**: performs non-linear transformation according to the SBOX lookup table.

- **ShiftRows**: performs permutation on the data on a byte level.
- **MixColumns**: performs a multiplication with a constant matrix in Galois Field GF(2^8).
- **AddRoundKey**: performs the addition of the Round Key.

The instructions are implemented for accelerating the encryption and key-schedule functions (Appendix D) of the AES block cipher.

- **aes32esi rd, rs1, rs2, bs** : AES final round encryption instruction, 5.20.

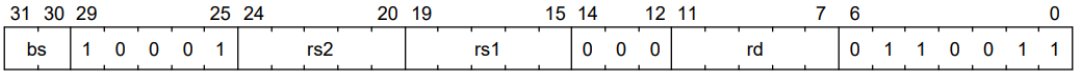

Figure 5.20: Aes32esi Encoding [7]

- **aes32esmi rd, rs1, rs2, bs** : AES middle round encryption instruction, 5.21.

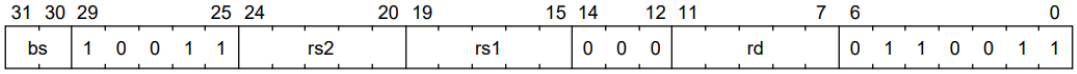

Figure 5.21: Aes32esmi Encoding [7]

5.1.5 Zknd - NIST Suite: AES Decryption

The decryption algorithm has the same structure as the encryption. However, each layer is replaced by its inverse and so the Byte Substitution layer becomes the Inv Byte Substitution layer, the ShiftRows layer becomes the Inv ShiftRows layer, and the MixColumn layer becomes Inv MixColumn layer. Furthermore the order of the round keys is reversed. These instructions are implemented for accelerating the decryption and key-schedule functions of the AES block cipher.

- **aes32dsi rd, rs1, rs2, bs** : AES final round decryption instruction, 5.22.

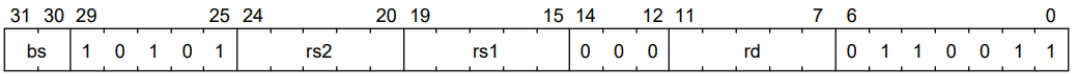

Figure 5.22: Aes32dsi Encoding [7]

- **aes32dsmi rd, rs1, rs2, bs** : AES middle round decryption instruction, 5.23.

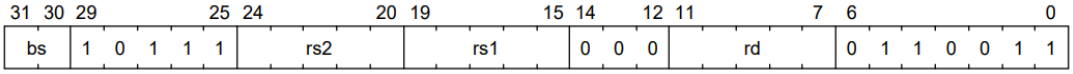

Figure 5.23: Aes32dsmi Encoding [7]

5.2 Instruction Hardware Implementation

5.2.1 Brev8

The brev8 instruction reverses the bits in each byte of rs source register in rd destination register. The Figure shows 5.24 the implementation.

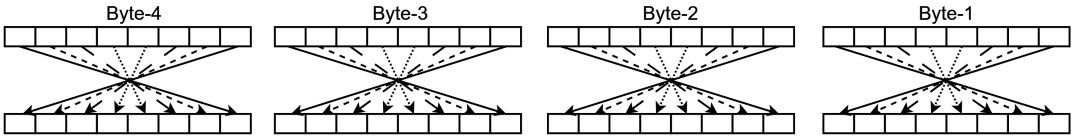

Figure 5.24: Brev8 implementation

5.2.2 Pack

The pack instruction packs the lower halves of rs1 and rs2 into rd, with rs1 in the lower half and rs2 in the upper half. The implementation is showed in Figure 5.25.

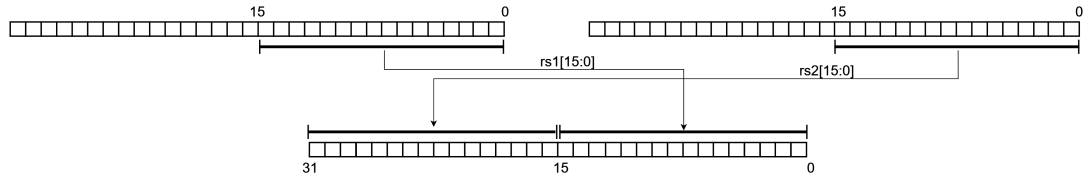

Figure 5.25: Pack implementation

5.2.3 Packh

The packh instruction packs the least-significant bytes of rs1 and rs2 into the first and second least-significant bytes of rd, respectively. Zero extending the rest of rd. The implementation is showed in Figure 5.26.

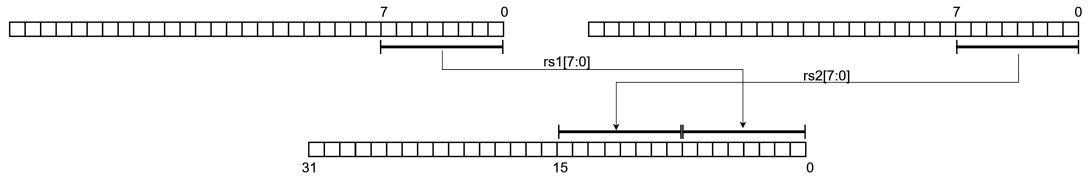

Figure 5.26: Packh implementation

5.2.4 Zip

This instruction places bits in the low half of the source register into the even bit positions of the destination, and bits in the high half of the source register into the odd bit positions of the destination. The implementation is showed in Figure 5.27.

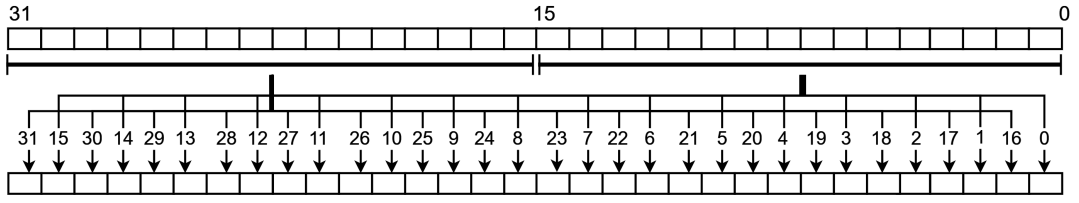

Figure 5.27: Zip implementation

5.2.5 Unzip

This instruction is the inverse of the zip instruction. So It places the even bits of the source register into the low half of the destination, and the odd bits of the source into the high bits of the destination. The implementation is showed in Figure 5.28.

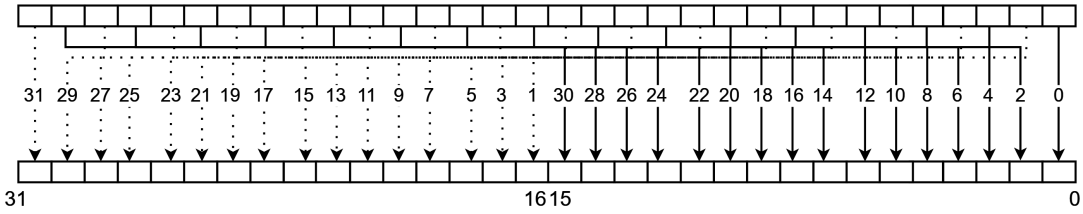

Figure 5.28: Unzip implementation

5.2.6 Xperm8

This instruction performs the permutation of the bytes of rs1 following the indexes taken from each byte of rs2. If the index is out of bound, zeros are inserted. The instruction is implemented through multiple multiplexers. For each of them, the selection signal is the i-th byte of rs2 (i = 0 to 3) that chooses one of the bytes of rs1 or 8-bit zero and saves it to the i-th byte of rd. The implementation is showed in Figure 5.29.

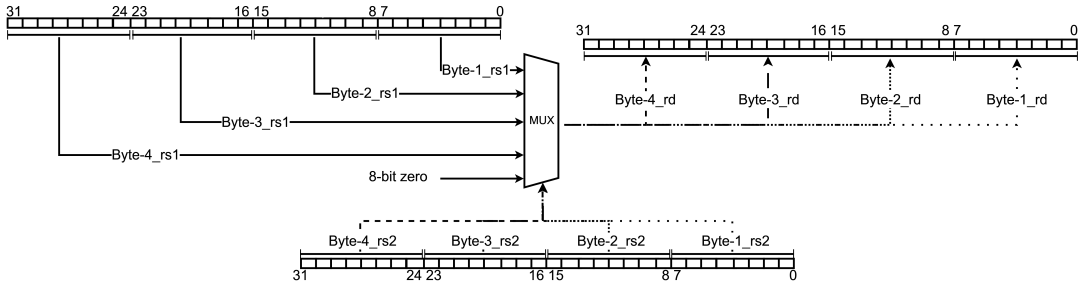

Figure 5.29: Xperm8 implementation

5.2.7 Xperm4

This instruction performs the permutation of the nibbles of rs1 following the indexes taken from each nibble of rs2. If the index is out of bound, zeros are inserted. The instruction is implemented through multiple multiplexers. For each of them, the selection signal is the i-th nibble of rs2 (i = 0 to 7) that chooses one of the nibbles of rs1 or 4-bit zero and saves it to the i-th nibble of rd. The implementation is showed in Figure 5.30.

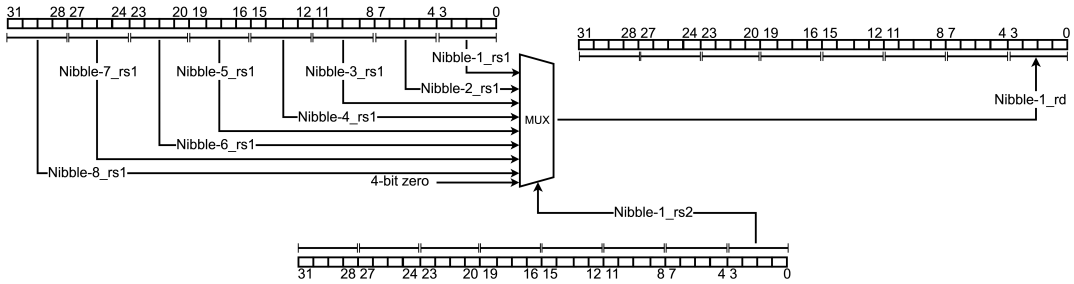

Figure 5.30: Xperm4 implementation

5.2.8 Sha256sig0

The Sha512sig0 implements the Sigma0 transformation function(5.5), as used in the SHA2-256 hash function.

$$\mathrm{ROR}(rs1, 7) \oplus \mathrm{ROR}(rs1, 18) \oplus \mathrm{SRL}(rs1, 3) \tag{5.11}$$

The source register rs1 on 32 bits is taken, the proper rotations and shift, on 32 bits, are applied and then the results are Xor-ed. The implementation is showed in Figure 5.31.

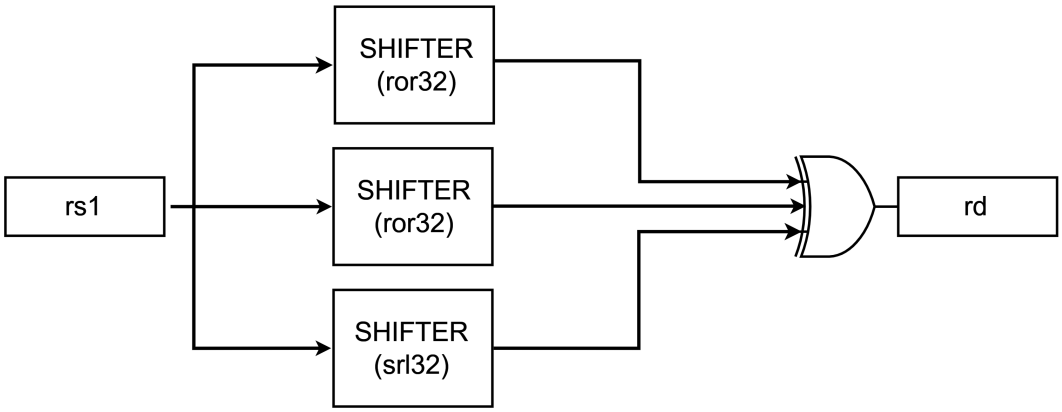

Figure 5.31: Sha256sig0 implementation

5.2.9 Sha256sig1

The Sha512sig1 implements the Sigma1 transformation function(5.6), as used in the SHA2-256 hash function.

$$\mathrm{ROR}(rs1, 17) \oplus \mathrm{ROR}(rs1, 19) \oplus \mathrm{SRL}(rs1, 10) \tag{5.12}$$

The process involves taking the 32-bit source register rs1, applying the necessary rotations and shifts, on 32 bits, and then combining the results using the XOR operation. The implementation is the same as the previous one, and It is showed in Figure 5.32.

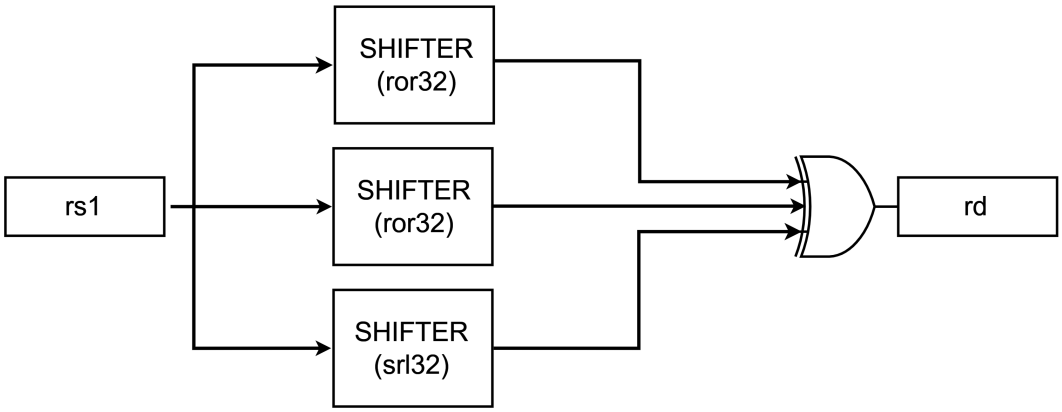

Figure 5.32: Sha256sig1 implementation

5.2.10 Sha256sum0

The Sha512sum0 implements the Sum0 transformation function(5.3), as used in the SHA2-256 hash function.

$$\text{ROR}(rs1, 2) \oplus \text{ROR}(rs1, 13) \oplus \text{ROR}(rs1, 22) \tag{5.13}$$

The process involves taking the 32-bit source register rs1, applying the necessary rotations, on 32 bits, and then combining the results using the XOR operation. The implementation is showed in Figure 5.33.

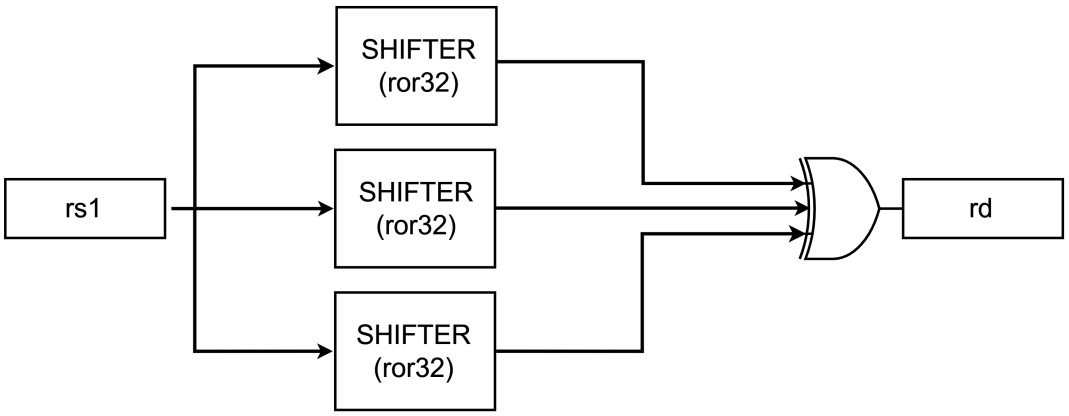

Figure 5.33: Sha256sum0 implementation

5.2.11 Sha256sum1

The Sha512sum0 implements the Sum1 transformation function as used in the SHA2-256 hash logical function 5.4.

$$\text{ROR}(rs1, 6) \oplus \text{ROR}(rs1, 11) \oplus \text{ROR}(rs1, 25) \tag{5.14}$$

The process involves taking the 32-bit source register rs1, applying the necessary rotations, on 32 bits, and then combining the results using the XOR operation. The implementation is showed in Figure 5.34.

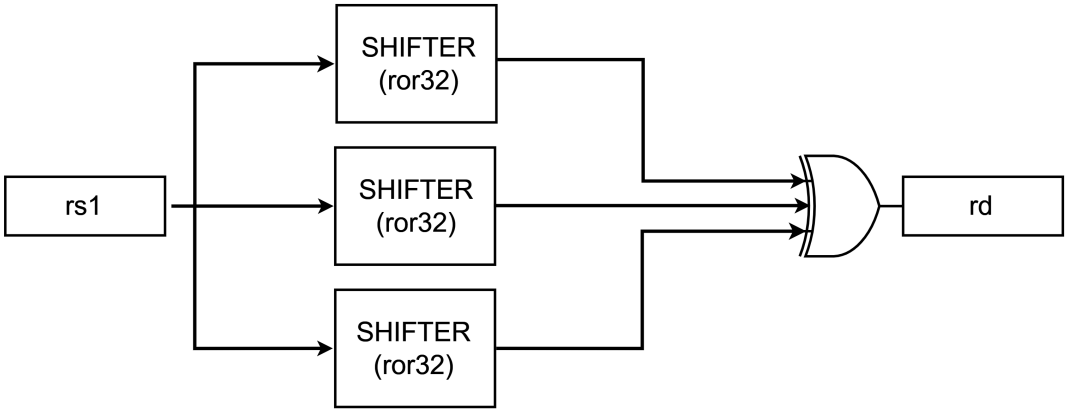

Figure 5.34: Sha256sum1 implementation

5.2.12 Sha256

All the Sha256 intructions have been implemented in a single path to have an efficient hardware implementation. As all the instructions are based on shifters, the main structure is always based on them. To ensure the proper execution of rotations and shifts, two signals have been introduced to guide each shifter in making the appropriate operation. The signals are:

- cmd : This 1-bit signal is used to know if the instruction concerns the sigma function or the sum function.

- n : This 1-bit signal is used to know if the function selected by the cmd signal refers to the 0 function or the 1 function.

So the process always involves taking the 32-bit register rs1, applying the 32-bit rotations or shifts, depending on cmd and n, and then combining the results using the XOR operation. The implementation is showed in Figure 5.35.

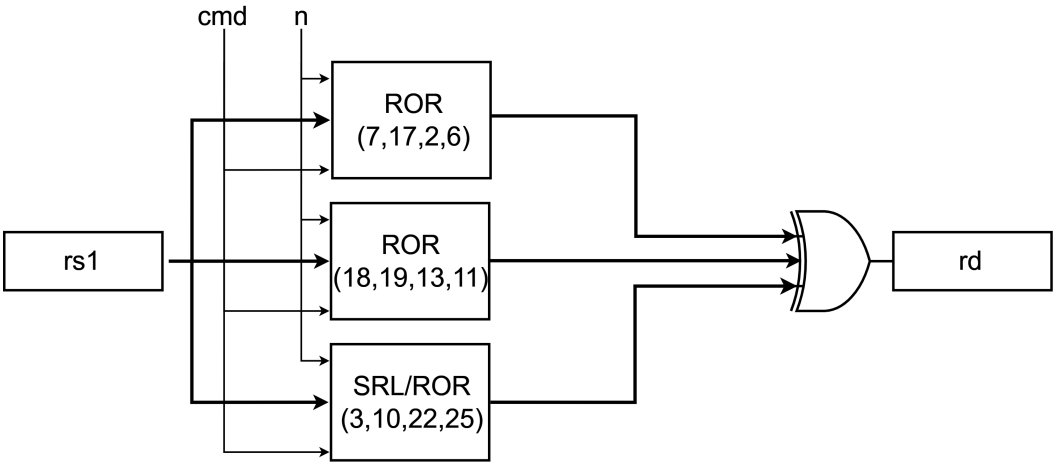

Figure 5.35: Sha256 implementation

5.2.13 Sha512sig0h

The sha512sig0h instruction implements the high half of the Sigma0 transformation within the SHA-512 hash function, while the sha512sig0l instruction complements this by executing the other part of the Sigma0 function. These instructions work together to implement the complete Sigma0 function (5.9).

$$\text{SRL}(rs1, 1) \oplus \text{SRL}(rs1, 7) \oplus \text{SRL}(rs1, 8) \oplus \text{SLL}(rs2, 31) \oplus \text{SLL}(rs2, 24) \quad (5.15)$$

As previously mentioned, since the instruction is designed for a 64-bit operation within a 32-bit architecture, it requires the use of two 32-bit registers. This function executes various shifts to obtain the accurate result for the high half of the Sigma 0 function. Both registers are used because the specified shifts may involve the movement of bits from the low part of rs2 (a0) to the high part of rs1 (a1). The implementation is showed in Figure 5.36.

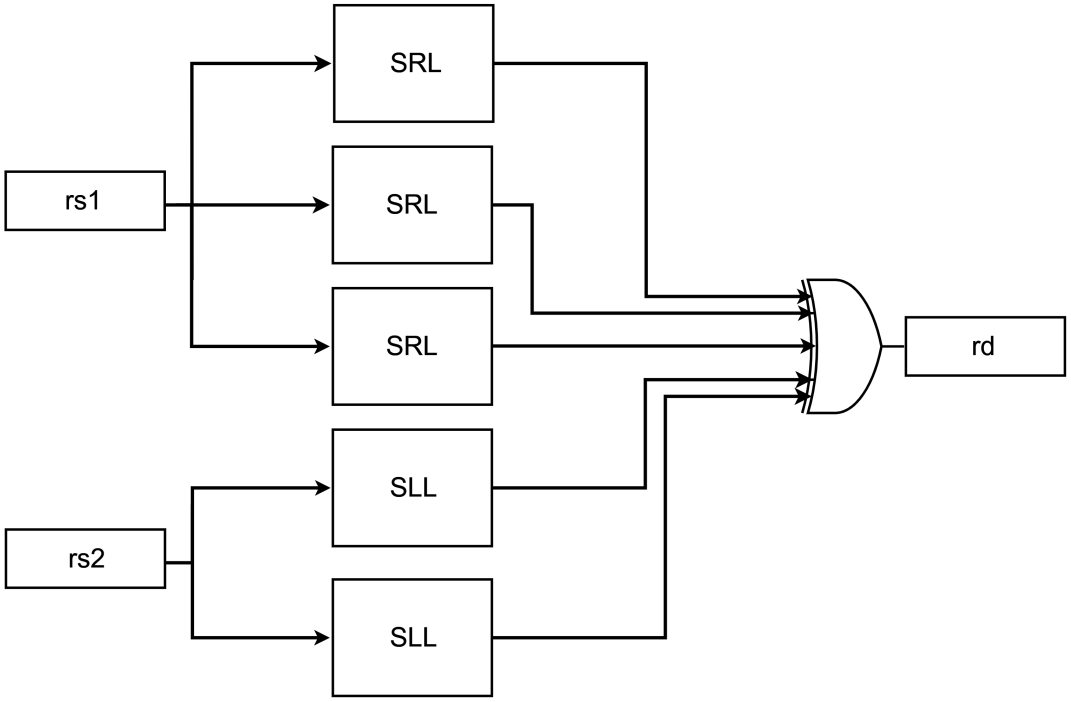

Figure 5.36: Sha512sig0h implementation

5.2.14 Sha512sig0l

The sha512sig0l instruction implements the low half of the Sigma0 transformation within the SHA2-512 hash function, and as said before, it works with the sha512sig0h to perform the Sigma0 function (5.9).

$$\mathrm{SRL}(rs1, 1) \oplus \mathrm{SRL}(rs1, 7) \oplus \mathrm{SRL}(rs1, 8) \oplus \mathrm{SLL}(rs2, 31) \oplus \mathrm{SLL}(rs2, 25) \oplus \mathrm{SLL}(rs2, 24) \quad (5.16)$$

Also in this case, since the instruction is designed for a 64-bit operation within a 32-bit architecture, two 32-bit registers are used. This function executes various shifts to obtain the accurate result for the low half of the Sigma0 function. Both registers are used because the specified shifts may involve the movement of bits from the high part of rs2 (a1) to the low part of rs1 (a0). The implementation is showed in Figure 5.37.

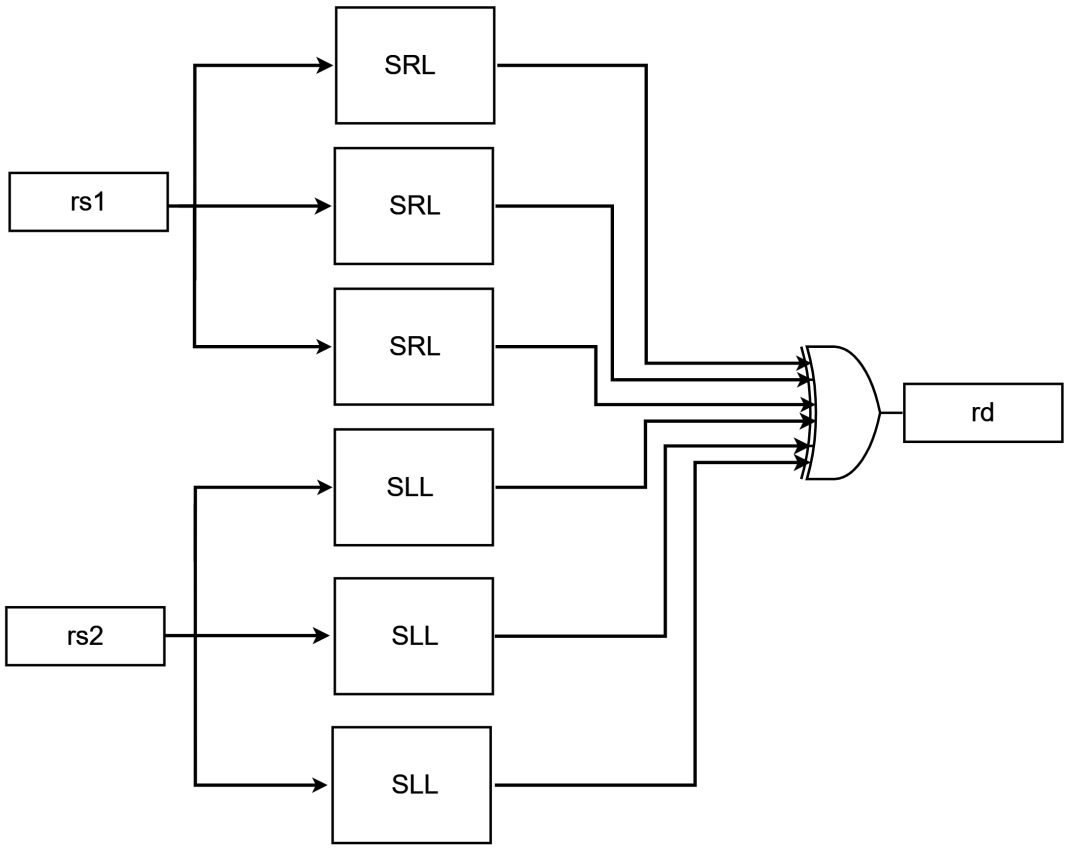

Figure 5.37: Sha512sig0l implementation

5.2.15 Sha512sig1h

The sha512sig1h instruction implements the high half of the Sigma1 transformation within the SHA-512 hash function, while the sha512sig1l instruction complements this by executing the other part of the Sigma1 function. These instructions work together to implement the complete Sigma0 function (5.10).

$$\text{SLL}(rs1, 3) \oplus \text{SRL}(rs1, 6) \oplus \text{SRL}(rs1, 19) \oplus \text{SRL}(rs2, 29) \oplus \text{SLL}(rs2, 13) \quad (5.17)$$

As previously mentioned, since the instruction is designed for a 64-bit operation within a 32-bit architecture, it requires the use of two 32-bit

registers. This function executes various shifts to obtain the accurate result for the high half of the Sigma 1 function. Both registers are used because the specified shifts may involve the movement of bits from the low part of rs2 (a0) to the high part of rs1 (a1). The implementation is showed in Figure 5.38.

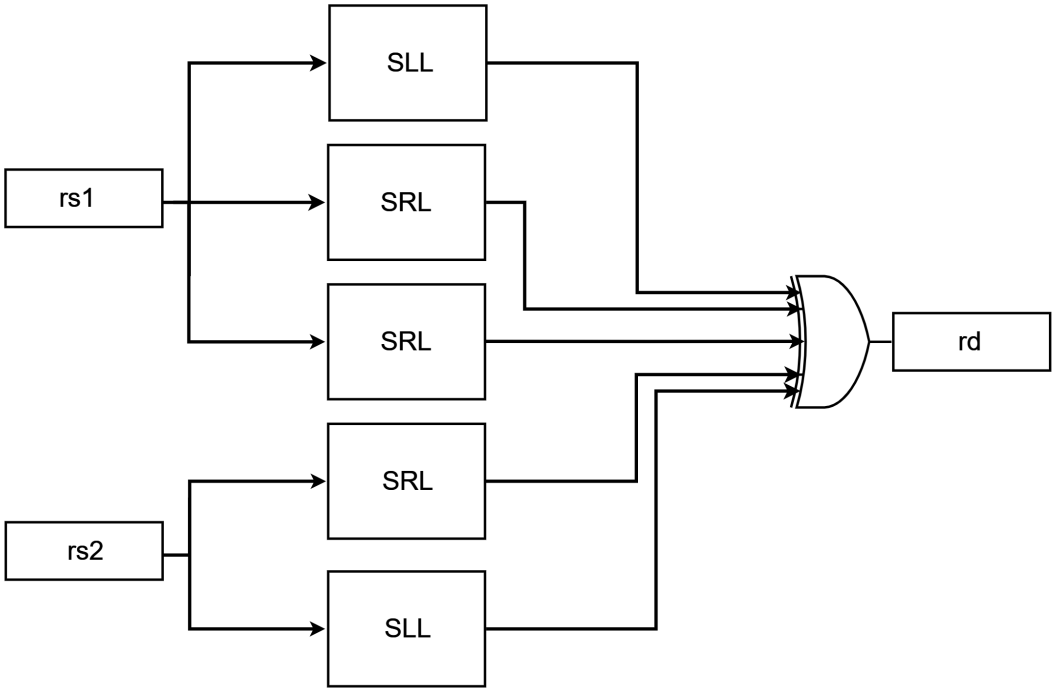

Figure 5.38: Sha512sig1h implementation

5.2.16 Sha512sig1l

The sha512sig1l instruction implements the low half of the Sigma1 transformation within the SHA2-512 hash function, and as said before, it works with the sha512sig1h to perform the Sigma1 function (5.10).

$$\mathrm{SLL}(rs1, 3) \oplus \mathrm{SRL}(rs1, 6) \oplus \mathrm{SRL}(rs1, 19) \oplus \mathrm{SRL}(rs2, 29) \oplus \mathrm{SLL}(rs2, 26) \oplus \mathrm{SLL}(rs2, 13) \quad (5.18)$$

Also in this case, since the instruction is designed for a 64-bit operation within a 32-bit architecture, two 32-bit registers are used. This function executes various shifts to obtain the accurate result for the low half

of the Sigma1 function. Both registers are used because the specified shifts may involve the movement of bits from the high part of rs2 (a1) to the low part of rs1 (a0). The implementation is showed in Figure 5.39.

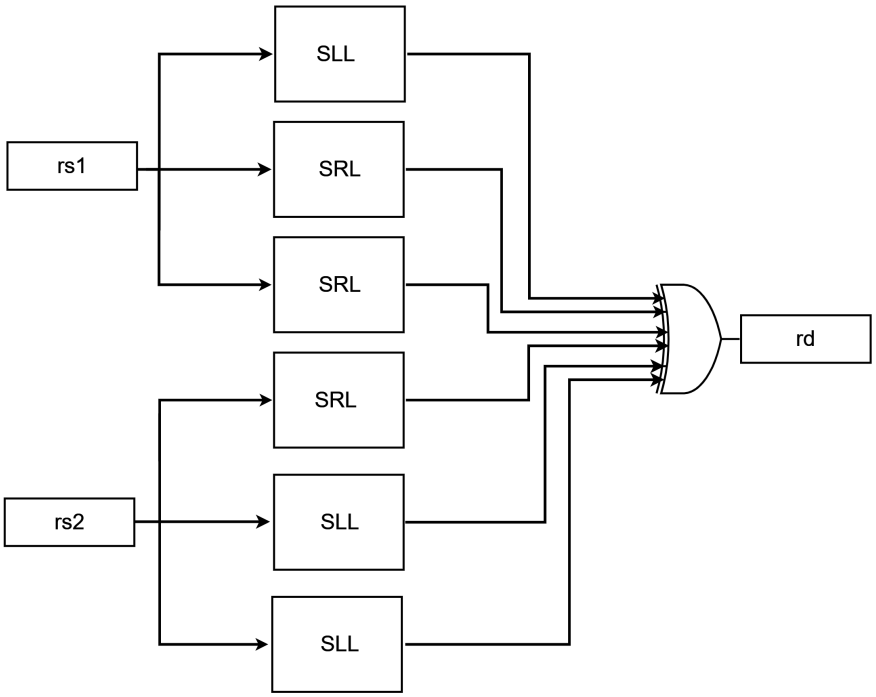

Figure 5.39: Sha512sig1l implementation

5.2.17 Sha512sum0r

The sha512sum0r instruction implements the Sum0 transformation within the SHA2-512 hash function, and as said in the specifications, it is repeated two times to implement the Sum0 function (5.7).

$$\mathrm{SLL}(rs1, 25) \oplus \mathrm{SLL}(rs1, 30) \oplus \mathrm{SRL}(rs1, 28) \oplus \mathrm{SRL}(rs2, 7) \oplus \mathrm{SRL}(rs2, 2) \oplus \mathrm{SLL}(rs2, 4) \quad (5.19)$$

Also in this case, since the instruction is designed for a 64-bit operation within a 32-bit architecture, two 32-bit registers are used. This function executes various shifts to obtain the accurate result considering that the instruction is called two times. The two registers are inverted to

imply a rotation by 32 bits. The implementation is showed in Figure 5.40.

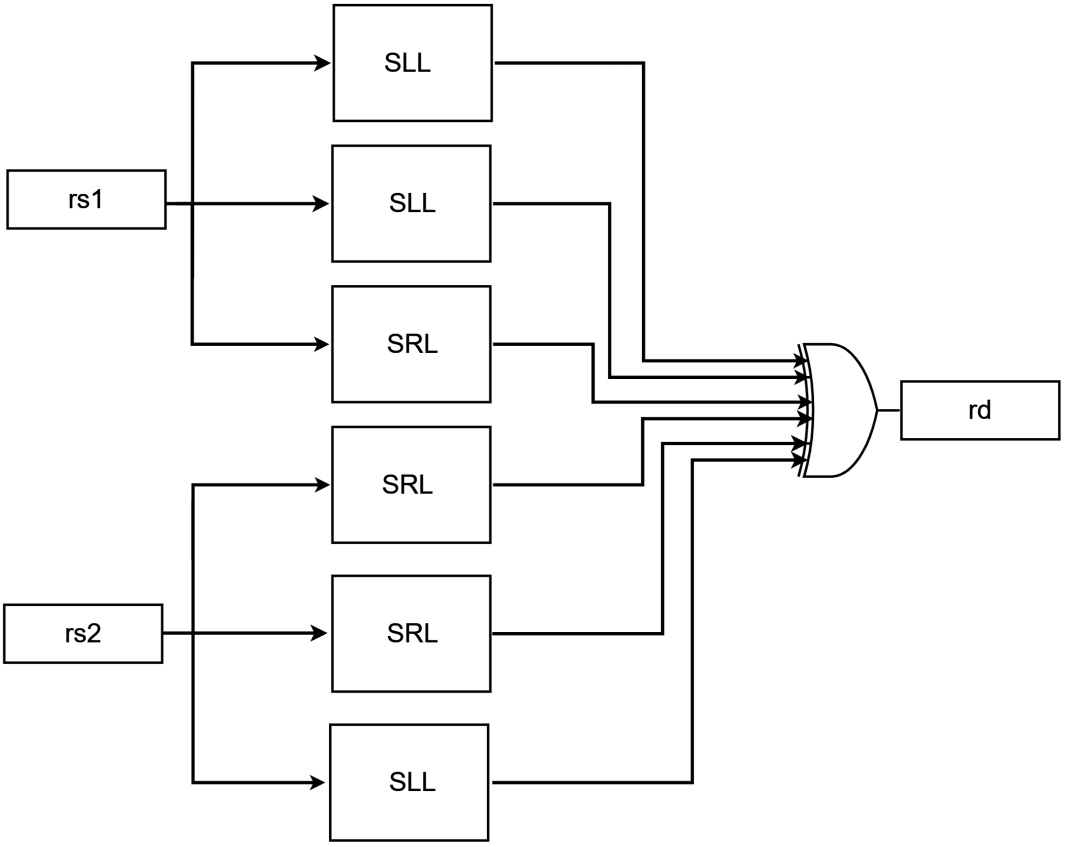

Figure 5.40: Sha512sum0r implementation

5.2.18 Sha512sum1r

The sha512sum0r instruction implements the Sum1 transformation within the SHA2-512 hash function, and as said in the specifications, it is repeated two times to implement the Sum1 function (5.8).

$$\mathrm{SLL}(rs1, 23) \oplus \mathrm{SRL}(rs1, 14) \oplus \mathrm{SRL}(rs1, 18) \oplus \mathrm{SRL}(rs2, 9) \oplus \mathrm{SLL}(rs2, 18) \oplus \mathrm{SLL}(rs2, 14) \quad (5.20)$$

Also in this case, since the instruction is designed for a 64-bit operation within a 32-bit architecture, two 32-bit registers are used. This function

executes various shifts to obtain the accurate result considering that the instruction is called two times. The two registers are inverted to imply a rotation by 32 bits. The implementation is showed in Figure 5.41.

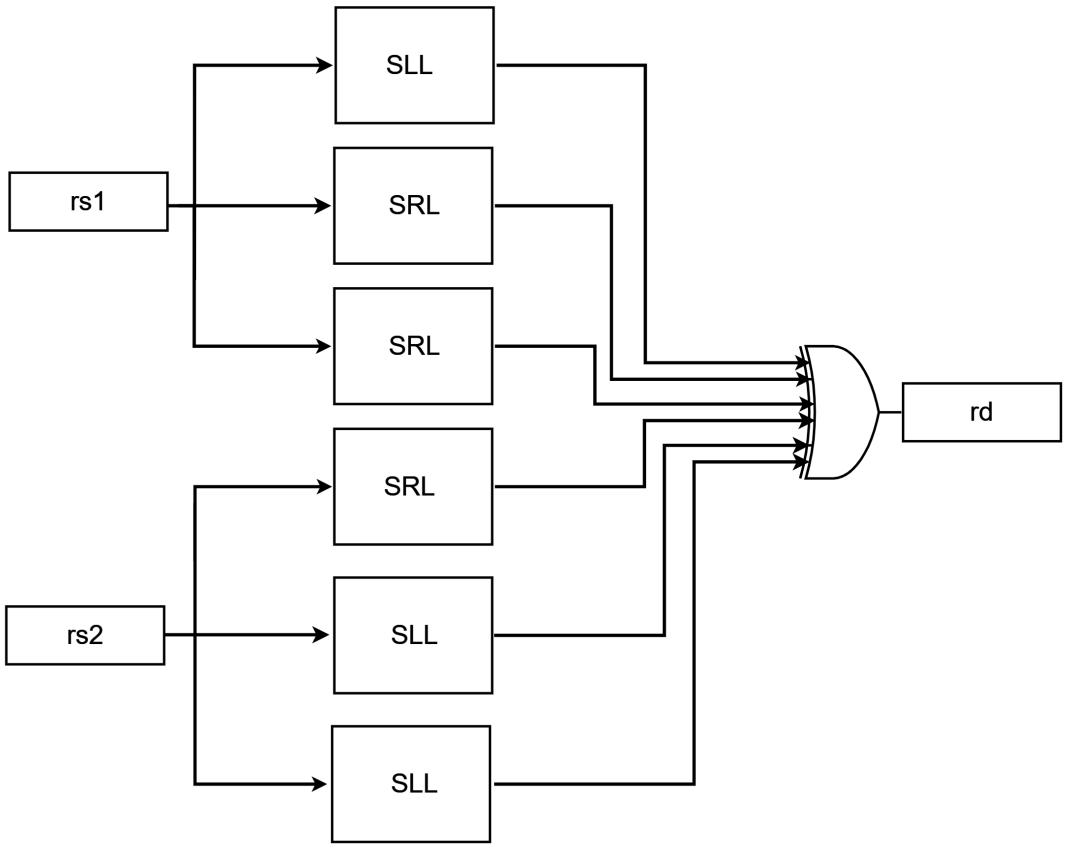

Figure 5.41: Sha512sum1r implementation

5.2.19 Sha512

All the Sha512 intructions have been implemented in a single path to have an efficient hardware implementation. As all the instructions are based on shifters, the main structure is always based on them. To ensure the proper execution of shifts, three signals have been introduced to guide each shifter in making the appropriate operation. The signals are:

- l : This 2-bit signal is used to know if the instruction concerns the high half or the low half. It is also used to know if It is a sum.
- f : This 1-bit signal is used to know which function has to be implemented. So if It is a Sigma o a Sum function.
- n : This 1-bit signal is used to know if the function selected by the f signal refers to the 0 function or the 1 function.

To be even more efficient, a tree of XOR has been implemented to not have a big XOR with 6 inputs and 1 output. So the process always involves taking the 2 32-bit registers rs1 and rs2, applying the 32-bit shifts, depending on f, l and n, and then combining the results using the XOR tree. The implementation is showed in Figure 5.42.

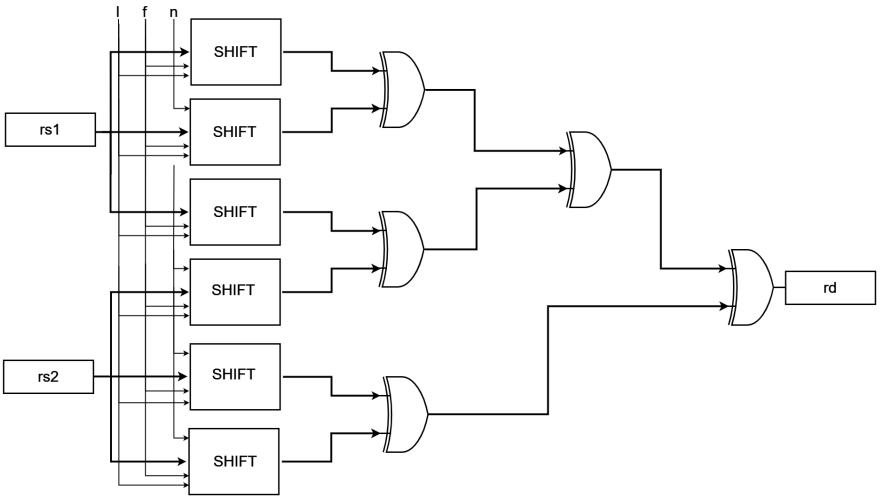

Figure 5.42: Sha512 implementation

5.2.20 aes32esi

The aes32esi instruction selects one of the 4 bytes from the rs2 register based on the 2-bit bs value. It then applies the forward AES SBOX transformation (SubBytes) to the selected byte, saves the byte back to its original position, and sets the remaining bits to zero (ShiftRows). Subsequently, the outcome is rotated left by bs values and XORed

with the contents of the rs1 register (AddRoundKey) and stored in the destination register rd. The implementation is showed in Figure 5.43.

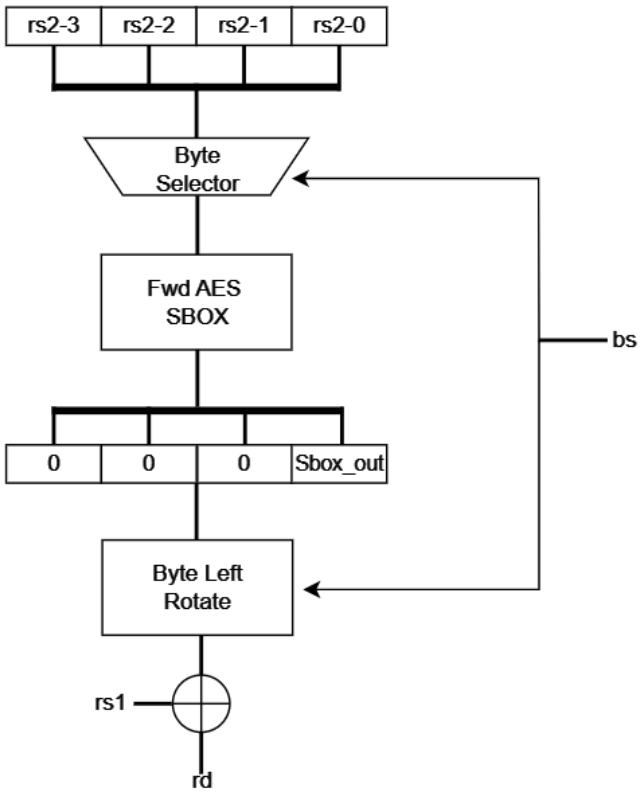

Figure 5.43: aes32esi implementation

Forward Sbox - SubBytes

The AES substitution permutation network architecture utilizes the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$. This polynomial provides non-linear permutations for the cipher, and the AES S-Box is based on a non-linear boolean function that replaces an element of a finite field with its modular multiplicative inverse, denoted as x $\rightarrow$ (x^{-1}) in GF(2^8). The S-Box in AES has two modes of operation: SubByte and InvSubByte, making it invertible. The original AES specification defines the S-Box as the multiplicative inverse in the field GF(2^8), followed by an affine transformation showed in Figure 5.44.

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \\ s_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Figure 5.44: Forward AES Affine Transformation

The software approach often involves using a lookup table (Appendix D), but in this case, the implementation follows the Boyar-Peralta method [25]. Since AES operations are based on XOR and AND operations, which are equivalent to addition and multiplication in GF(2), circuits for cryptographic functions often use this basis. In GF(2), negation corresponds to x + 1, but for accurate gate-count, XNOR gates are used instead of negation. The circuits implemented by Boyar-Peralta are divided into three components: top linear transformations, shared non-linear component, and bottom linear transformations. The S-Box expands an 8-bit input to 21 bits in a linear inner layer (top layer), utilizes the shared nonlinear 21-to-18 bit mapping as a middle layer, and compresses 18 bits back to 8 bits in the outer layer (bottom layer). XOR and XNOR gates are considered "linear", while the shared nonlinear layer consists of XOR and AND gates only [25][24]. The new circuits have a depth of 16, with a size of 128 gates, and the shared component between the forward and inverse directions is of size 63. The Table 5.1 summarizes the algebraic gates count.

5.2.21 aes32esmi

The aes32esmi instruction operates by selecting one of the 4 bytes from the rs2 register based on the 2-bit bs value. It then applies the forward AES SBOX transformation. Additionally, the outcome is multiplied by

Component	In/Out	XOR	XNOR	AND	Total
Shared middle	21 → 18	30	-	34	64
Forward AES top	8 → 21	26	-	-	26
Forward AES bottom	18 → 8	34	4	-	38

Table 5.1: Boyar-Peralta algebraic gate counts Forward Sbox [24]

a column of the forward AES MixColumns matrix (Partial MixColumn). Finally, the bytes are rotated left according to bs, XORed with the contents of the rs1 register, and stored in the destination register RD. The implementation is showed in Figure 5.45.

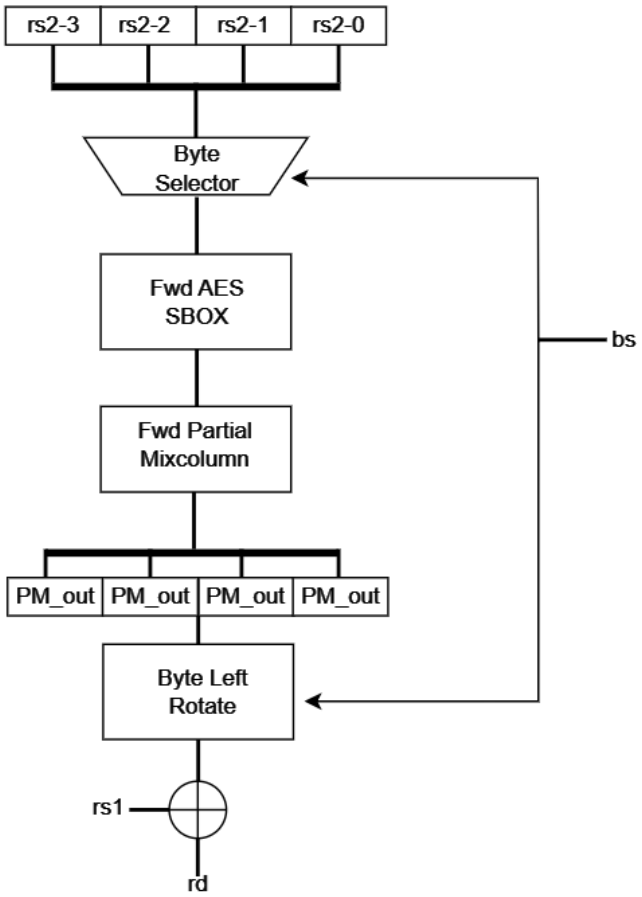

Figure 5.45: aes32esmi implementation

Forward Partial MixColumns

The MixColumn function is a linear algebra operation and serves as the primary source of diffusion in AES. In this operation, each column of bytes is treated as a four-term polynomial $b(x) = b_3x^3 + b_2x^2 + b_1x + b_0$, where each byte represents an element in the Galois Field GF(2^8). The coefficients are elements within the prime sub-field GF(2). Each column is multiplied with the fixed polynomial $a(x) = 3x^3 + x^2 + x + 2$ modulo $x^4 + 1$, the modulo operation is performed to ensuring that the result stays within the field. The matrix representation is $d_i = S \times b_i$, showed in Figure 5.46, where i is the column index for matrices d, b. The matrix S is a fixed coefficient matrix specific to the MixColumns operations.

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Figure 5.46: Forward MixColumns Matrix Representation

The vector-matrix multiplication forms the MixColumns operations as shown in Figure 5.47. It is important to note that each byte d_i and b_i is an 8-bit value representing an element from GF(2^8). The additions in the vector–matrix multiplication are GF(2^8) additions, involving simple bitwise XORs of the respective bytes. For the multiplication of the constants, the multiplications with the constants 01, 02 and 03 are explained below:

- Multiplying by 0x01 changes nothing, it is a multiplication by the identity element.
- Multiplying by 0x02 is equivalent to shifting all the digits of the byte by 1 position to the left (this is equivalent to multiplying the

corresponding polynomial by 21). In this case, it is possible that the result is out of the GF(2^8). To obtain a polynomial of degree 8, it is necessary to reduce by the characteristic polynomial of the AES $x^8 + x^4 + x^3 + x + 1$ which results in 9 bits represented as 0x11b in hexadecimal (0x1b as the values are on 8 bits).

- Multiplication by 0x03 corresponds to multiplication by 0x02, followed by addition of the original value (XOR operation in GF(2^8).

Since the instruction works on a 8-bit input with a 32-bit output, the implementation of the Partial MixColumns is focused on the first column of the MixColumns operations shown in Figure 5.47. So the Partial Mixcolumns performs the function on only the 8-bit b_0 input.

$$
\begin{aligned}
d_0 &= 2 \bullet b_0 \oplus 3 \bullet b_1 \oplus 1 \bullet b_2 \oplus 1 \bullet b_3 \\
d_1 &= 1 \bullet b_0 \oplus 2 \bullet b_1 \oplus 3 \bullet b_2 \oplus 1 \bullet b_3 \\
d_2 &= 1 \bullet b_0 \oplus 1 \bullet b_1 \oplus 2 \bullet b_2 \oplus 3 \bullet b_3 \\
d_3 &= 3 \bullet b_0 \oplus 1 \bullet b_1 \oplus 1 \bullet b_2 \oplus 2 \bullet b_3
\end{aligned}
$$

Figure 5.47: Forward MixColumns Operations

5.2.22 aes32dsi

The aes32dsi instruction selects one of the 4 bytes from the rs2 register based on the 2-bit bs value. It then applies the inverse AES SBOX transformation (Inverse SubBytes) to the selected byte, saves the byte back to its original position, and sets the remaining bits to zero (ShiftRows). Subsequently, the outcome is rotated left by bs values and XORed with the contents of the rs1 register (AddRoundKey) and stored in the destination register rd. The implementation is showed in Figure 5.48.

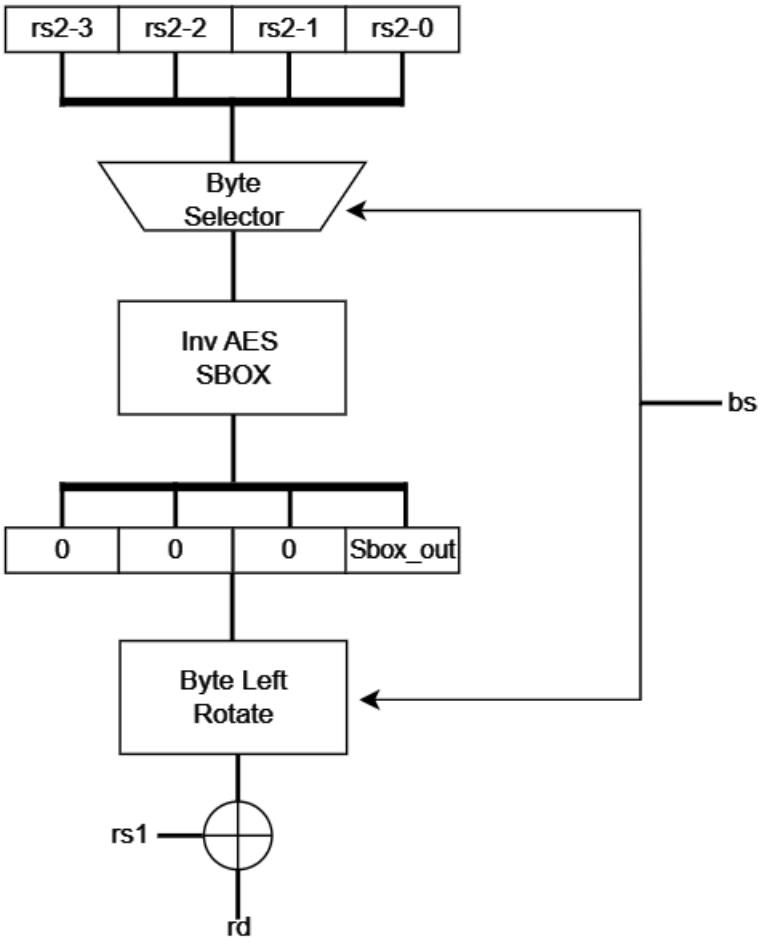

Figure 5.48: aes32dsi implementation

Inverse Sbox - Inverse SubBytes

Since the AES S-Box is invertible, It is possible to construct an inverse S-Box. So the inverse S-box is simply the Forward S-box reversed. It is calculated by first calculating the inverse affine transformation, in Figure 5.49, of the input value, followed by the multiplicative inverse.

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \\ s_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Figure 5.49: Inverse AES Affine Transformation

The software approach, as in the forward case, often involves using a lookup table (Appendix D), but, also for the inverse Sbox, the implementation follows the Boyar-Peralta method [25]. The circuits implemented by Boyar-Peralta are divided into three components: top linear transformations, shared non-linear component, and bottom linear transformations. The S-Box expands an 8-bit input to 21 bits in a linear inner layer (top layer), utilizes the shared nonlinear 21-to-18 bit mapping as a middle layer, and compresses 18 bits back to 8 bits in the outer layer (bottom layer). XOR and XNOR gates are considered "linear", while the shared nonlinear layer consists of XOR and AND gates only [25][24]. The new circuits have a depth of 16, with a size of 127 gates, and the shared component between the forward and inverse directions is of size 63. The Table 5.2 summarizes the algebraic gates count.

Component	In/Out	XOR	XNOR	AND	Total
Shared middle	21 → 18	30	-	34	64
Inverse AES top	8 → 21	16	10	-	26
Inverse AES bottom	18 → 8	37	-	-	37

Table 5.2: Boyar-Peralta algebraic gate counts Inverse Sbox [24]

5.2.23 aes32dsmi

The aes32dsmi instruction operates by selecting one of the 4 bytes from the rs2 register based on the 2-bit bs value. It then applies the inverse AES SBOX transformation. Additionally, the outcome is multiplied by a column of the inverse AES MixColumns matrix (Partial MixColumns). Finally, the bytes are rotated left according to bs, XORed with the contents of the rs1 register, and stored in the destination register RD. The implementation is showed in Figure 5.50.

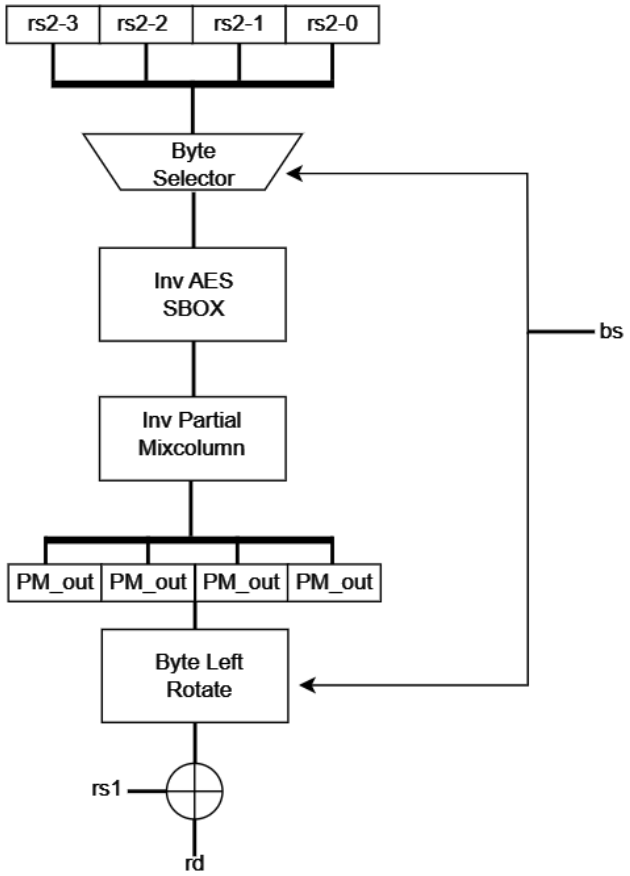

Figure 5.50: aes32dsmi implementation

Inverse Partial MixColumns

The Inverse MixColumns is implemented reversing the Forward MixColumns and following the same rules. In this case each column is multiplied with the fixed polynomial $a^{-1}(x) = 11x^3 + 13x^2 + 9x + 14$. The matrix representation is $b_i = C \times d_i$, showed in Figure 5.51, where i is the column index for matrices d, b. The matrix C is a fixed coefficient matrix specific to the MixColumn operations.

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix}$$

Figure 5.51: Inverse MixColumns Matrix Representation

The vector-matrix multiplication forms the MixColumns operations as shown in Figure 5.52. The additions in the vector–matrix multiplication are GF(2^8) additions, involving simple bitwise XORs of the respective bytes. For the multiplication of the constants, the multiplications with the constants 14, 9, 13 and 11 are explained below. For each multiplication (shift) performed, there is always a check to ensure that the resulting polynomial does not exceed the maximum degree of 8.

- Multiplying by 14 (0x0E) is equivalent to shift the initial byte by one position to the left, followed by a XOR with the initial byte shifted by 2 positions. Then, this result has to be XOR-ed with the initial byte shifted by 3 positions to the left.
- Multiplying by 9 (0x09) is equivalent to shift the initial byte by 3 position to the left, followed by a XOR with the initial byte.
- Multiplying by 13 (0x0D) is equivalent to shift the initial byte by 2 position to the left, followed by a XOR with the initial byte. Then,

this result has to be XOR-ed with the initial byte shifted by 3 positions to the left.

- Multiplying by 11 (0x0B) is equivalent to shift the initial byte by one position to the left, followed by a XOR with the initial byte. Then, this result has to be XOR-ed with the initial byte shifted by 3 positions to the left.

As before since the instruction works on a 8-bit input with a 32-bit output, the implementation of the Partial MixColumns is focused on the first column of the MixColumns operations shown in Figure 5.52. So the Partial Mixcolumns performs the function on only the 8-bit d_0 input.

$$
\begin{aligned}
b_0 &= 14 \bullet d_0 \oplus 11 \bullet d_1 \oplus 13 \bullet d_2 \oplus 9 \bullet d_3 \\
b_1 &= 9 \bullet d_0 \oplus 14 \bullet d_1 \oplus 11 \bullet d_2 \oplus 13 \bullet d_3 \\
b_2 &= 13 \bullet d_0 \oplus 9 \bullet d_1 \oplus 14 \bullet d_2 \oplus 11 \bullet d_3 \\
b_3 &= 11 \bullet d_0 \oplus 13 \bullet d_1 \oplus 9 \bullet d_2 \oplus 14 \bullet d_3
\end{aligned}
$$

Figure 5.52: Inverse MixColumns Operations

5.2.24 aes32

All the SHA-512 instructions have been implemented into a single path to achieve an efficient hardware implementation. Given the similarity in structure among these instructions, this unified structure has been maintained, with the addition of two signals to facilitate the execution of the required instruction.

- box : This 1-bit signal is utilized to determine whether the instruction relates to an encryption or decryption process. So if the SBox and Partial Mixcolumn are forward or inverse.
- mix : This 1-bit signal is used to determine whether the instruction relates to a final or middle round. So if the Partial Mixcolumn output has to be taken into account.

The implementation is showed in Figure 5.53.

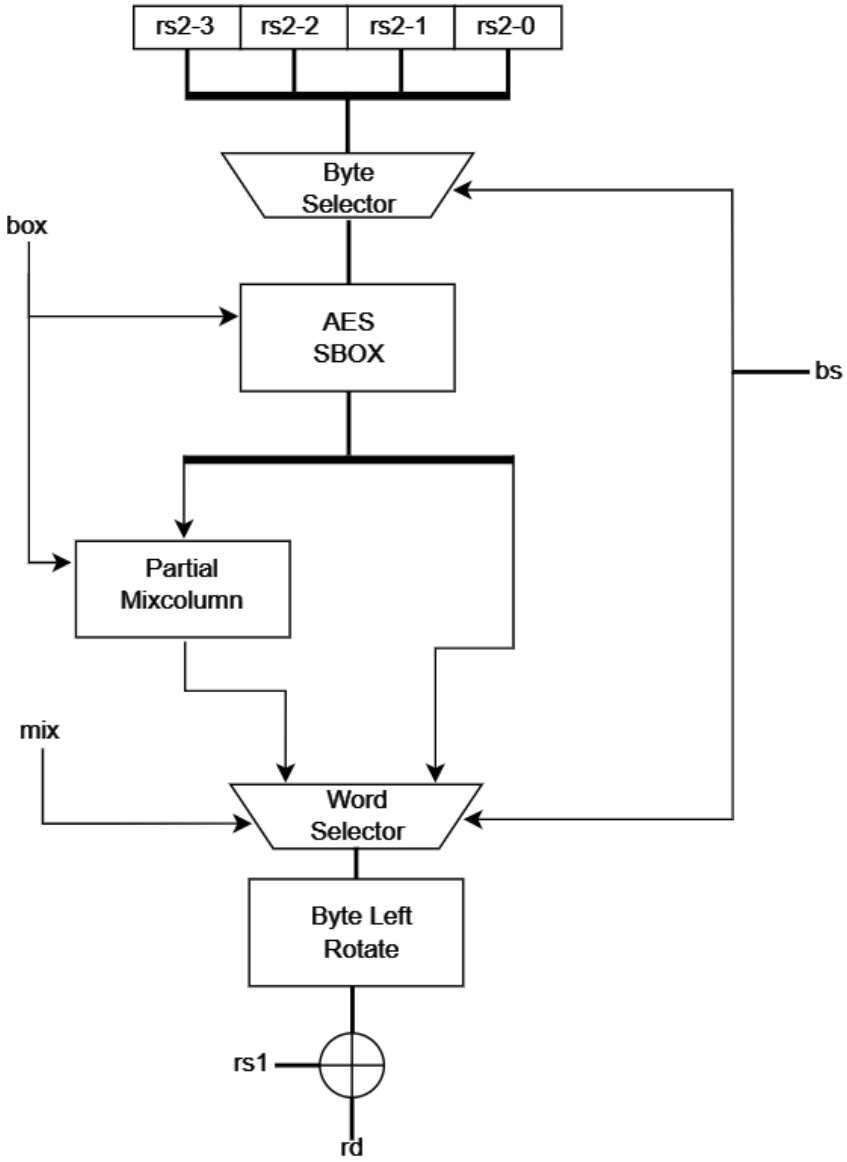

Figure 5.53: aes32 implementation

Chapter 6

Other Processor Crypto Extension

This chapter will explore how the Advanced Encryption Standard (AES) and Secure Hash Algorithm (SHA) are organized and implemented in processors from industry-leading companies such as Intel and ARM. Both Intel and ARM cryptographic ISAs utilize 128-bit SIMD registers, which may not be available on lower-end CPUs. While both Intel and ARM processors provide cryptographic instructions and features that can be leveraged on 32-bit architectures, the use of 128-bit SIMD registers is a common practice in modern cryptographic implementations. This practice aims to enhance performance, particularly for intricate cryptographic algorithms such as AES (Advanced Encryption Standard), which can leverage the parallel processing capabilities offered by wider registers. This overview is given to compare the RISC-V Crypto Extension with other processor implementations.

6.1 Intel Processors

Intel processors are designed to deliver efficient encryption, decryption and hashing functionalities. The AES and SHA instructions are supported into the Intel 64 and IA-32 instructions.

AES-NI Instructions

Intel AES-NI provides six instructions designed to accelerate symmetric block encryption/decryption using the Advanced Encryption Standard (AES) [26] [16]. These instructions target different stages of the AES encryption and decryption processes:

- **AESENC** and **AESENCLAST**: Target AES encryption rounds.
- **AESDEC** and **AESDECLAST**: Target AES decryption rounds using the Equivalent Inverse Cipher.
- **AESIMC**: Targets the Inverse MixColumn transformation primitive.
- **AESKEYGEN**: Targets the generation of round keys from the cipher key for AES encryption/decryption rounds.

The cryptography processing involves the following functions and transformations:

- MixColumns(): A byte-oriented 4x4 matrix transformation on the matrix representation of a 128-bit AES state.
- RotWord(): Performs a byte-wise cyclic permutation (rotate right in little-endian byte order) on a 32-bit AES word.
- ShiftRows(): A byte-oriented matrix transformation that cyclically shifts the last three rows of the state by different offsets to the left.
- SubBytes(): Applies a non-linear substitution table (S-BOX) on each byte of the 128-bit AES state.
- SubWord(): Produces an output AES word (four bytes) from the input word using a non-linear substitution table (S-BOX).
- InvMixColumns(): The inverse transformation of MixColumns().
- InvShiftRows(): The inverse transformation of ShiftRows(), cyclically shifting the last three rows of the state by different offsets to the right.

- InvSubBytes(): The inverse transformation of SubBytes().

SHA-NI Instructions

SHA extensions provide a set of instructions that target the acceleration of the Secure Hash Algorithm (SHA), specifically the SHA-1 and SHA-256 variants. The SHA-512 is not supported as stand alone instructions [27].

- **SHA1MSG1**: Perform an intermediate calculation for the next four SHA1 message dwords from the previous message dwords.
- **SHA1MSG2**: Perform the final calculation for the next four SHA1 message dwords from the intermediate message dwords.
- **SHA1NEXTE**: Calculate SHA1 state E after four rounds.
- **SHA1RNDS4**: Perform four rounds of SHA1 operations.
- **SHA256MSG1**: Perform an intermediate calculation for the next four SHA256 message dwords.
- **SHA256MSG2**: Perform the final calculation for the next four SHA256 message dwords.
- **SHA256RNDS2**: Perform two rounds of SHA256 operations.

6.2 Arm Processors

Armv8-A architecture extension provides the AES instructions to support AES encryption and decryption, SHA-1 and SHA-256 instructions. The SHA-512 has been implemented in the Armv8.2.

AES Instructions

AES-128 is implemented through 2 instructions for the encryption and 2 instructions for the decryption [28] [16].

- **AESE** : Perform a single round of the AddRoundKey(), SubBytes(), and ShiftRows() transformations.
- **AESMC** : Perform a single round of the MixColumns() transformation.
- **AESD** : Perform a single round of the AddRoundKey(), InvSubBytes(), and InvShiftRows() transformations.
- **AESMC** : Perform a single round of the InvMixColumns() transformation.

SHA Instructions

The ARM crypto extension includes a set of specialized instructions for accelerating SHA-1, SHA-256, and SHA-512 hashing operations [28].

- **SHA1C**: SHA1 hash update (choose).
- **SHA1H**: SHA1 fixed rotate.
- **SHA1M**: SHA1 hash update (majority).
- **SHA1P**: SHA1 hash update (parity).
- **SHA1SU0**: SHA1 schedule update 0.
- **SHA1SU1**: SHA1 schedule update 1
- **SHA256H**: SHA256 hash update (part 2).
- **SHA256H2**: SHA256 hash update (part 1).
- **SHA256SU0**: SHA256 schedule update 0.
- **SHA256SU1**: SHA256 schedule update 1.
- **SHA512H**: SHA512 Hash update part combines the sigma1 and chi functions of two iterations of the SHA512 computation.

- **SHA512H2**: SHA512 Hash update part 2 combines the sigma0 and majority functions of two iterations of the SHA512 computation.
- **SHA512SU0**: SHA512 Schedule Update 0 combines the gamma0 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block.
- **SHA512SU1**: SHA512 Schedule Update 1 combines the gamma1 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block.

Chapter 7

Simulation and Formal Verification

Dynamic verification involves testing the Scalar crypto extension implementation by executing it with various inputs to observe its behavior. This process aims to validate the functionality and correctness of the design through simulation and hardware testing. Key aspects of dynamic verification include:

- Test Bench: Developing comprehensive test benches to verify the functionality and performance through simulation and emulation.
- Compliance Test Suite: Creating a test suite that ensures the design complies with industry standards and specifications.
- Self-Checked Random Tests: Implementing self-checking mechanisms and random tests to validate the design under various scenarios and inputs.
- Applications: Testing the design in real-world applications to assess its practical usability and effectiveness.
- Certitude Qualification: Qualifying the design for certitude by ensuring its reliability, security, and compliance with standards and requirements.

The git repository of the verification team contains C tests called "sanity" for dynamic verification and files in tcl or verilog format contain assertions or conditions for formal proofs. Locally, you can launch a set of sanity tests from the Poker, the verification cockpit, interface and monitor failed or passed tests. Poker shows, in the case of a failed test, the error message which makes debugging easier. You can then launch the test with the ASIP commands and consult the Log files which contain more detailed information about execution and errors. The test is based on multiple executions:

- Native gcc Compilation and Simulation: Native execution with GCC refers to compiling and running a program directly on the host machine's.

- Compilation with Chess:

  - CAS Simulation: The cycle-accurate simulation (CAS) provides a detailed view of the hardware design's behavior at the cycle level.

  - RTL Simulation: The RTL simulator is used to simulate and verify the functionality of a hardware design described at the register-transfer level.

The traces given by the three different simulations are compared to validate the processor's behavior or to identify any discrepancies. The organization of the execution of the test is showed in Figure 7.1.

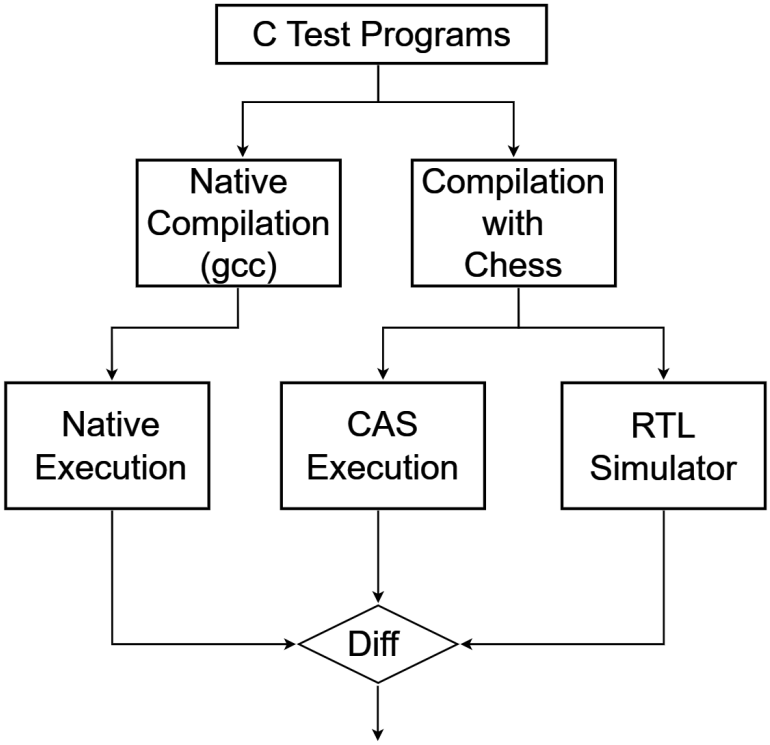

Figure 7.1: Test Flow

For observing the signals of the simulation, the unified graphical debugging environment for Cadence simulator Simvision, in Figure 7.2, tool is able to point to the Verilog code and observe the microarchitecture produced. To debug from the C code of the test, we can use ChessDE (from Synopsis), in Figure 7.3, which allows to set break points at each instruction, to observe the code Assembler produces, the state of registers, memory and pipeline.

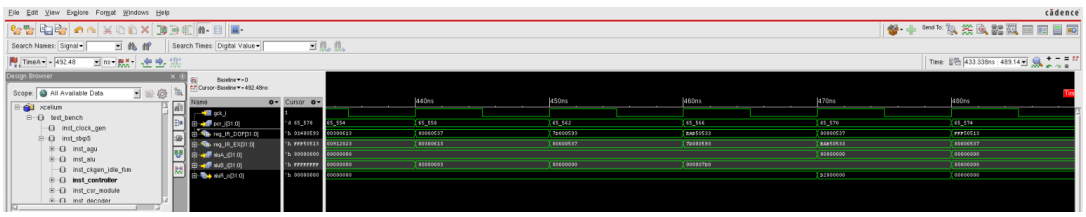

Figure 7.2: SimVision Debug Environment

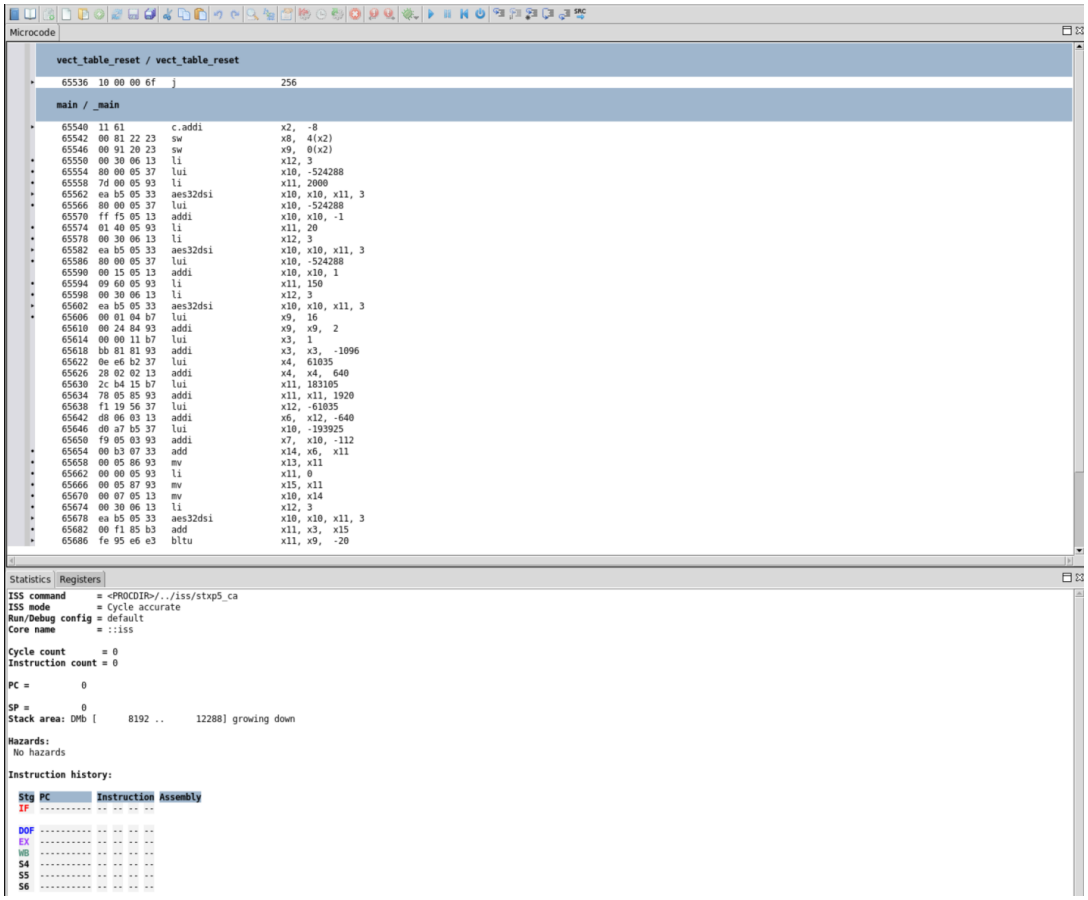

Figure 7.3: ChessDE's Graphical Debugger

Formal verification is part of the verification flow and It involves mathematically proving the correctness of the Scalar crypto extension design using formal methods and tools. This approach provides rigorous verification by exhaustively analyzing the design's behavior against specified (expressed as SystemVerilog Assertion (SVA)) properties. Key aspects of formal verification for the Scalar crypto extension include:

- Lint, RTL Static Signoff: Performing linting and static signoff checks to identify and rectify design issues at the RTL level before further verification.
- Standard Protocols: Verifying standard protocols to ensure interoperability and compliance.

- Properties: Defining formal properties and constraints to be verified using formal methods to ensure correctness and adherence to specifications.

- Coverage Unreachability: Analyzing coverage to identify unreachable code and ensure that all parts of the design are exercised during verification. The coverage is measured in two different ways:

  – Functional Coverage: It is used to measure the completeness of testing based on the functionality specified in the design specification. The Figure 7.4 shows a possible result.

⊟ COVERAGE_FROM_RISK	100%	596 / 596 (100%)
⊞ C.alu_rrr_ar_instr.op	100%	10 / 10 (100%)
⊞ C.alu_rrr_ar_instr.rd	100%	16 / 16 (100%)
⊞ C.st_mpy_rrr_instr.rd	100%	16 / 16 (100%)
⊞ C.div_instr.op	100%	2 / 2 (100%)
⊞ C.div_instr.rd	100%	16 / 16 (100%)
⊞ C.majOP	100%	3 / 3 (100%)
⊞ C.alu_rris_ar_instr.op	100%	6 / 6 (100%)
⊞ C.alu_rris_ar_instr.rd	100%	16 / 16 (100%)
⊞ C.alu_rri_sh_instr.op	100%	3 / 3 (100%)
⊞ C.alu_rri_sh_instr.rd	100%	16 / 16 (100%)
⊞ C.majOP_IMM	100%	2 / 2 (100%)
⊞ C.load_instr.op	100%	5 / 5 (100%)
⊞ C.load_instr.rd	100%	16 / 16 (100%)
⊞ C.majLOAD	100%	1 / 1 (100%)
⊞ C.store_instr.op	100%	3 / 3 (100%)
⊞ C.majSTORE	100%	1 / 1 (100%)
⊞ C.br_instr.op	100%	6 / 6 (100%)
⊞ C.majBRANCH	100%	1 / 1 (100%)
⊞ C.jal_instr.rd	100%	16 / 16 (100%)

Figure 7.4: Functional Coverage Analysis

  – RTL coverage: It focuses on verifying the coverage of the design at the register transfer level to ensure that the RTL code has been exhaustively exercised during simulation. The Figure 7.5 shows a possible result.

Name	Overall Average Grade	Overall Covered
⊟ stxp5_0	97.56%	78106 / 89060 (87.7%)
inst_alu	100%	235 / 235 (100%)
inst_lx	100%	105 / 105 (100%)
inst_agu	99.89%	351 / 352 (99.72%)
inst_mul_step	100%	748 / 748 (100%)
⊞ inst_dma_icn	93.68%	668 / 756 (88.36%)
inst_icn	95.63%	2114 / 2181 (96.93%)
inst_pma_core_req	99.34%	801 / 815 (98.28%)
inst_pma_dma_req	95.77%	264 / 286 (92.31%)

Figure 7.5: RTL Coverage Analysis

- X Propagation: Addressing X propagation issues to eliminate

unknown states in the design and ensure deterministic behavior during operation.

The Jasper tool allows it to turn formal proofs and observe failed properties. The CAD Tools used for Simulation&Fornal Verification are summarised in Table 7.1.

Synopsys	**Cadence**
ASIP Designer (ChessDE)	Xcelium Logic Simulator
Design Compiler	SimVision Debug Environment
RTLArchitect	Jasper

Table 7.1: CAD Tools

Chapter 8

Synthesis

The synthesis is run through Fekit that means FrontEnd Kit, and this is a set of tools (linter, synthesis, DFT insertion, ATPG generation,...) that has been developed by ST to have a consistent way to use ST Libraries. Among those different tools, the synthesis tool rely on Synopsys Synthesis tool (dc_shell) to convert RTL code into optimized gate level representation (netlist) using a 40 nm technology given a standard cell library (ST library CMOS40LP) and certain design constraints (Frequency at 100 MHz). The synthesis flow is showed in Figure 8.1.

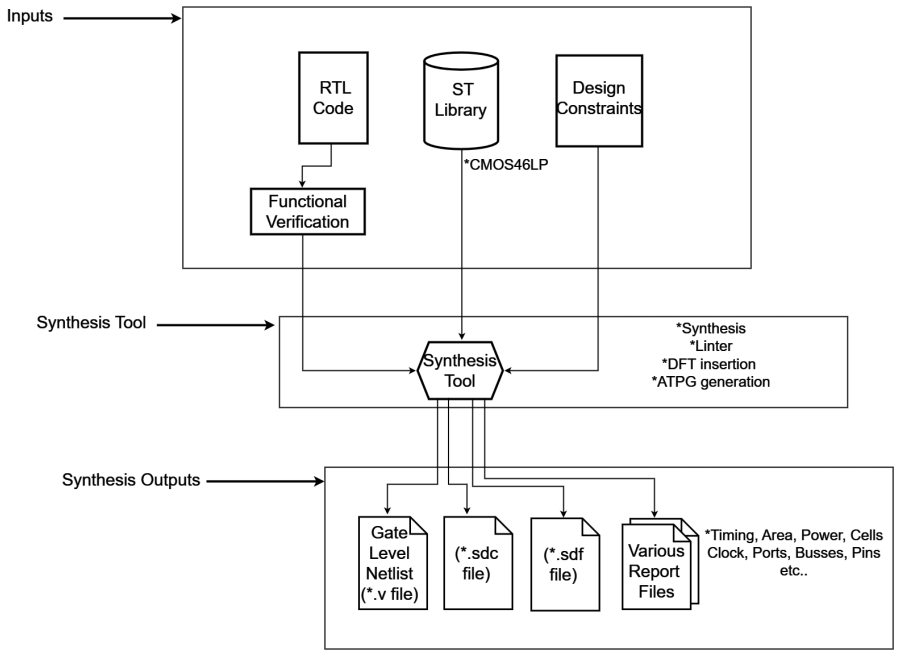

Figure 8.1: Synthesis Flow

8.1 Area Analysis

The main focus on the synthesis outputs is the area report, which lists the units of the processor in terms of area. The area is provided in micrometers (µm) but more importantly in the number of gates. The number of gates is a metric that consists in expressing the area of the processor in the amount of the selected reference gate. The reference gate chosen is the NAND2X2 gate with area 0.6804 μm^2. The area of the Arithmetic Logic Unit (ALU) is the main focus to analyze the impact of the extensions since all the instructions are directly implemented there. The baseline area, representing the ALU without the extensions, is set at 3.1 kilogates (kGates). By analyzing the area reports and specifically focusing on the ALU area to make comparison, the impact of the extensions on the overall design complexity and gate count can be evaluated.

Zbkb

The Zbkb extension has a minor impact on the ALU area, resulting in an increase of 0.1 kGates. Due to the small increase, no significant optimizations are necessary.

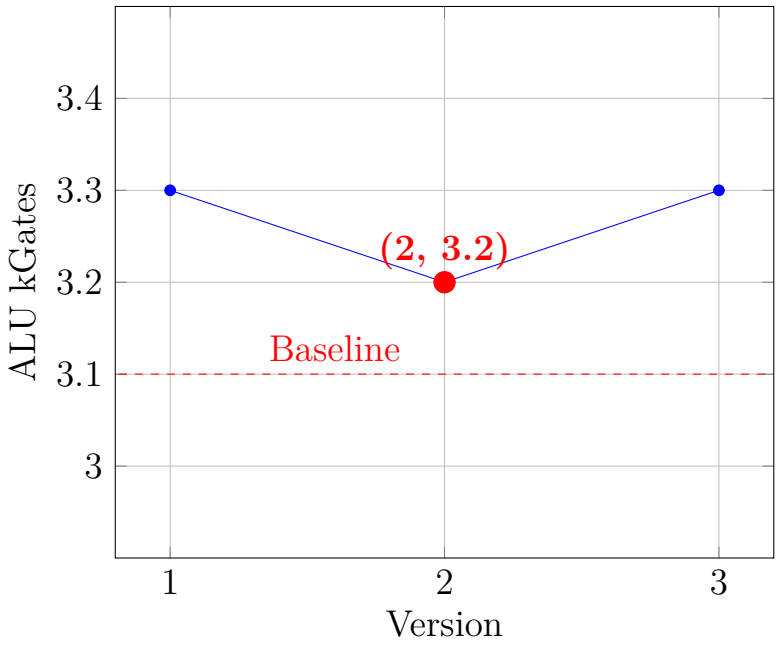

Figure 8.2: Zbkb Area Analysis

Zbkx

The Zbkx extension impacts with an increase in area of 0.5 kGates. In this scenario, different implementations have been tried, following three different principles:

- Shift : Using shift operator, the correct byte or nibble of rs1 was taken and put in the correct position of rd.
- One function : Trying to implement the two instructions in one function, always using shift operator.
- Hardwired: Directly by hand the two registers have been divided

in bytes or nibbles and with a switch-case on rs2, the correct byte or nibble of rs1 is put in the correct position of rd.

The Hardwired principle is the most optimized approach and therefore, it is the one selected as seen in Figure 5.29 and in Figure 5.30.

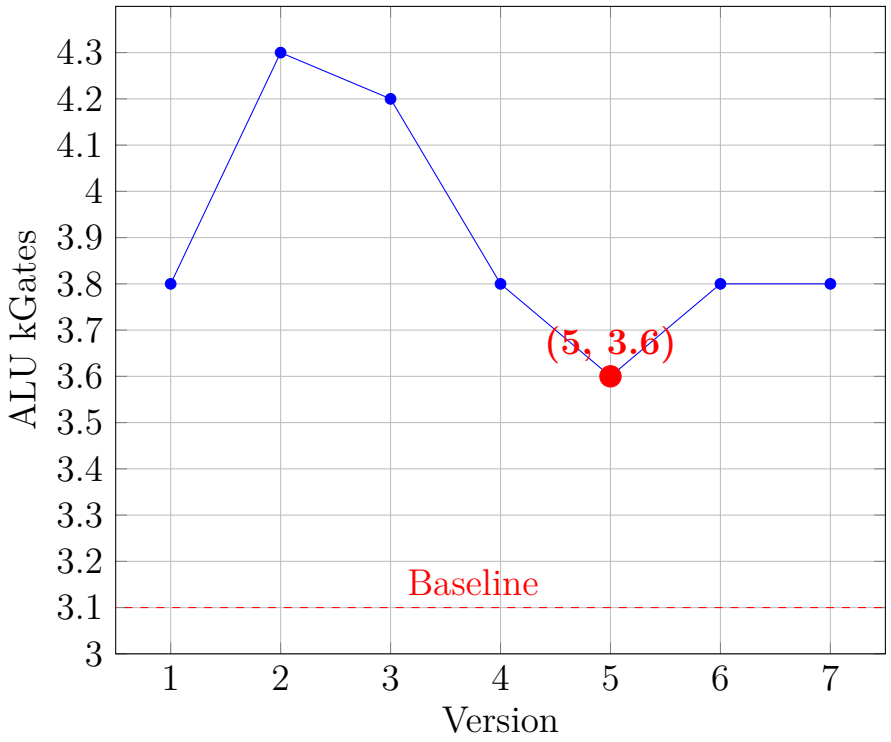

Figure 8.3: Zbkx Area Analysis

Zknh

The Zknh extension results in an increase in area of 1.0 kGates. In the first versions were based on one function for each instruction, leading to a significant increase in area. Subsequently, two functions were created for sha256 and sha512, respectively. Initially, the functions were written by shifting the operator and XORing with the previous one, using also operations already used in other instructions. Then, in the chosen implementation, all operators are prepared first, followed by the XOR-tree operation. This approach has been adopted to optimize

the area efficiently as it is shown in Figure 5.35 and in Figure 5.42. It is important to note that all shifts are constant shifts, avoiding the need for a barrel shifter and using simple XORs.

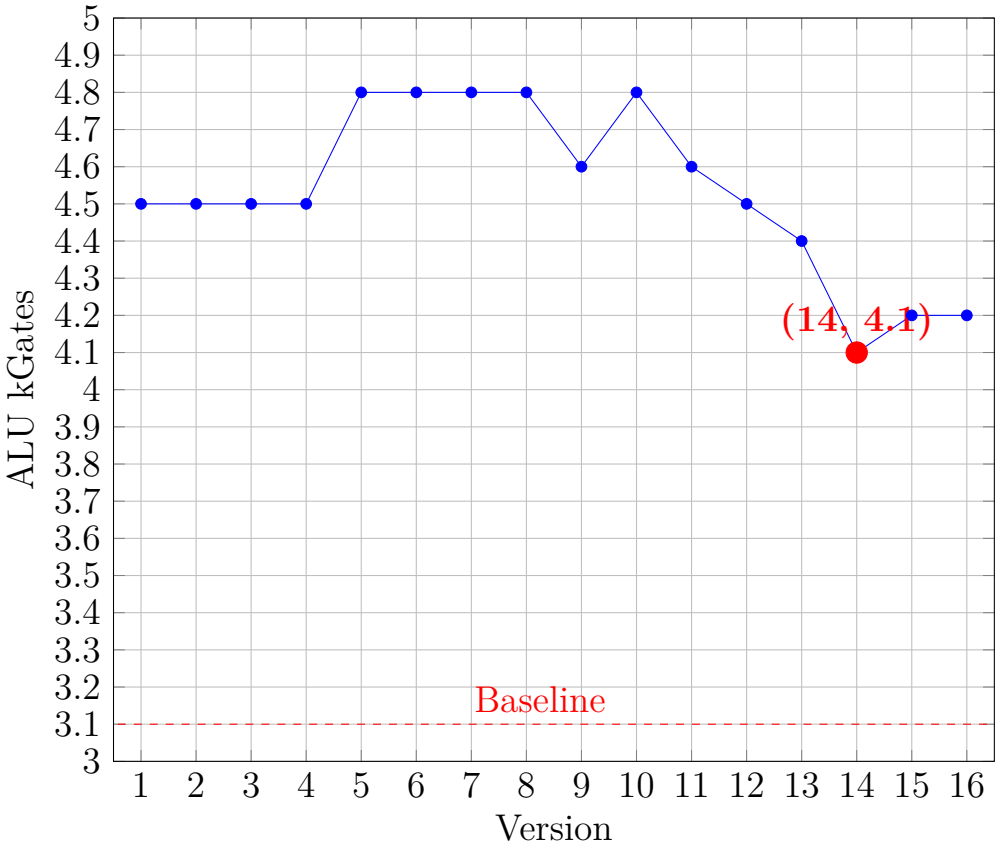

Figure 8.4: Zknh Area Analysis

Zkne+Zknd

The Zkne extension and the Zknd extension were implemented initially with one function for each instruction resulting, as before, in a significant increase of area. Then, a single main function has been implemented to perform all the different steps as shown in Figure 5.53. To make a comparison also a hardware implementation of the look-up table was also done, resulting in the maximum observed increase in area. The two extensions can be also implemented alone if needed. In this case

the impact is equal to +0.7 kGates for each extension.

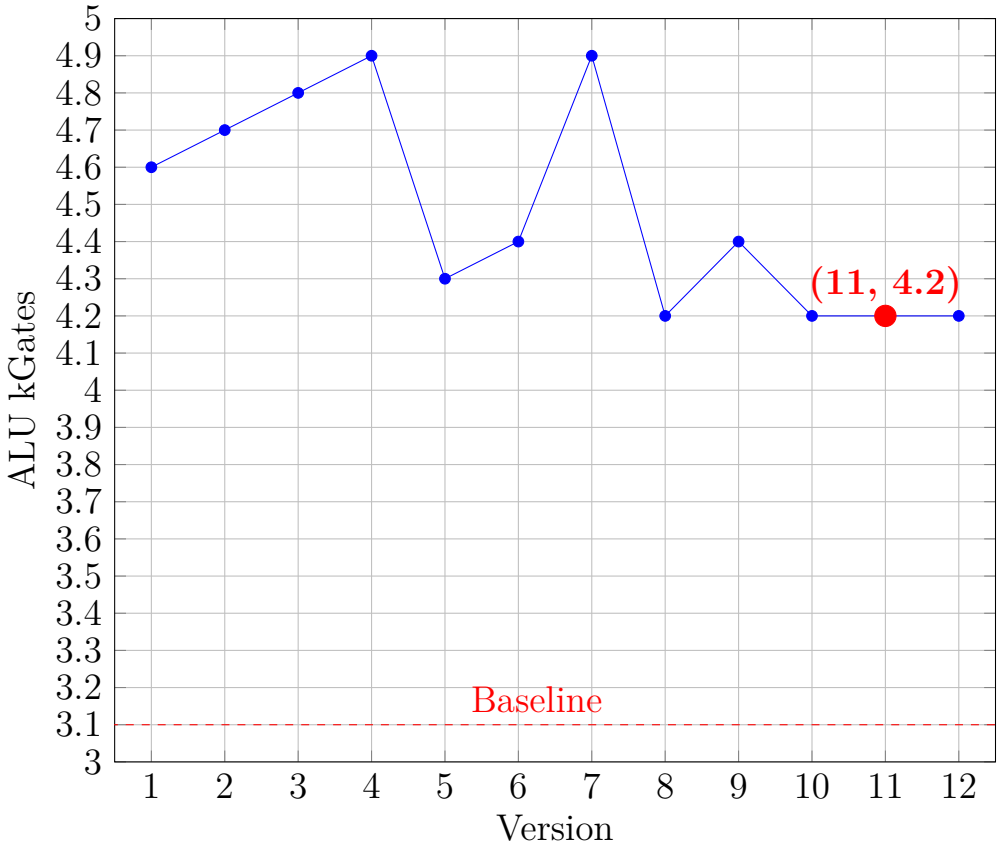

Figure 8.5: Zkne+Zknd Area Analysis

A key consideration was the handling of the immediate value bs for the ALU. Two approaches were considered:

- DOF stage approach: the value bs is taken in DOF and driven in EXE stage through a register. This approach may offer a timing advantage but could potentially lead to increased area usage.

- EXE stage approach: the value bs is taken directly in EXE since the encoding of the instruction is pipelined. While this approach may have a timing disadvantage, it could offer benefits in terms of area efficiency.

Given that the addition of a 2-bit register has minimal impact on the processor's area, the solution of taking bs in the DOF stage has been chosen.

Chapter 9

Benchmarks

In this chapter, the performance evaluation of two algorithms will be presented. The analysis will cover both software and hardware implementations. The software implementation utilizes only the base instruction set, while the hardware implementation uses the base instruction set and also a subset of cryptography instructions. The algorithms are the AES algorithm and the SHA-256 algorithm.

9.1 AES algorithm

The AES-128 algorithm has been run starting form a specific benchmark taken from Embech open source benchmark suite [29]. The code is based on 3 main parts:

- 2 main functions:
  - `_nettle_aes_encrypt` : Function to perform the Encryption. It is composed of a main loop that performs all the middle rounds and the final round. In addition there is key management and the results is stored. Figure 9.1 shows how it works.

```
_nettle_aes_encrypt (unsigned rounds, const uint32_t * keys,
                     const struct aes_table *T,
                     size_t length, uint8_t * dst, const uint8_t * src)
{
  FOR_BLOCKS (length, dst, src, AES_BLOCK_SIZE)
  {
    uint32_t w0, w1, w2, w3;    /* working ciphertext */
    uint32_t t0, t1, t2, t3;
    unsigned i;

    /* Get clear text, using little-endian byte order.
     * Also XOR with the first subkey. */

    w0 = LE_READ_UINT32 (src) ^ keys[0];
    w1 = LE_READ_UINT32 (src + 4) ^ keys[1];
    w2 = LE_READ_UINT32 (src + 8) ^ keys[2];
    w3 = LE_READ_UINT32 (src + 12) ^ keys[3];


    for (i = 1; i < rounds; i++)
      {
        t0 = AES_ROUND (T, w0, w1, w2, w3, keys[4 * i]);
        t1 = AES_ROUND (T, w1, w2, w3, w0, keys[4 * i + 1]);
        t2 = AES_ROUND (T, w2, w3, w0, w1, keys[4 * i + 2]);
        t3 = AES_ROUND (T, w3, w0, w1, w2, keys[4 * i + 3]);

        /* We could unroll the loop twice, to avoid these
           assignments. If all eight variables fit in registers,
           that should give a slight speedup. */
        w0 = t0;
        w1 = t1;
        w2 = t2;
        w3 = t3;
      }

    /* Final round */

    t0 = AES_FINAL_ROUND (T, w0, w1, w2, w3, keys[4 * i]);
    t1 = AES_FINAL_ROUND (T, w1, w2, w3, w0, keys[4 * i + 1]);
    t2 = AES_FINAL_ROUND (T, w2, w3, w0, w1, keys[4 * i + 2]);
    t3 = AES_FINAL_ROUND (T, w3, w0, w1, w2, keys[4 * i + 3]);


    LE_WRITE_UINT32 (dst, t0);
    LE_WRITE_UINT32 (dst + 4, t1);
    LE_WRITE_UINT32 (dst + 8, t2);
    LE_WRITE_UINT32 (dst + 12, t3);
  }
}
```

Figure 9.1: Encryption Function

– `_nettle_aes_decrypt` : Function to perform the Decryption. It is based on the same structure of the encrypt function and so main loop (middle rounds) plus a final round. As before, there is key management and the results is stored Figure 9.2 shows how it works.

```
_nettle_aes_decrypt (unsigned rounds, const uint32_t * keys,
                     const struct aes_table *T,
                     size_t length, uint8_t * dst, const uint8_t * src)
{
  FOR_BLOCKS (length, dst, src, AES_BLOCK_SIZE)
  {
    uint32_t w0, w1, w2, w3;    /* working ciphertext */
    uint32_t t0, t1, t2, t3;
    unsigned i;

    /* Get clear text, using little-endian byte order.
     * Also XOR with the first subkey. */

    w0 = LE_READ_UINT32 (src) ^ keys[0];
    w1 = LE_READ_UINT32 (src + 4) ^ keys[1];
    w2 = LE_READ_UINT32 (src + 8) ^ keys[2];
    w3 = LE_READ_UINT32 (src + 12) ^ keys[3];

    for (i = 1; i < rounds; i++)

    {
        t0 = AES_ROUND_DEC (T, w0, w3, w2, w1, keys[4 * i]);
        t1 = AES_ROUND_DEC (T, w1, w0, w3, w2, keys[4 * i + 1]);
        t2 = AES_ROUND_DEC (T, w2, w1, w0, w3, keys[4 * i + 2]);
        t3 = AES_ROUND_DEC (T, w3, w2, w1, w0, keys[4 * i + 3]);
        /* We could unroll the loop twice, to avoid these
           assignments. If all eight variables fit in registers,
           that should give a slight speedup. */
        w0 = t0;
        w1 = t1;
        w2 = t2;
        w3 = t3;
    }

    t0 = AES_FINAL_ROUND_DEC (T, w0, w3, w2, w1, keys[4 * i]);
    t1 = AES_FINAL_ROUND_DEC (T, w1, w0, w3, w2, keys[4 * i + 1]);
    t2 = AES_FINAL_ROUND_DEC (T, w2, w1, w0, w3, keys[4 * i + 2]);
    t3 = AES_FINAL_ROUND_DEC (T, w3, w2, w1, w0, keys[4 * i + 3]);

    LE_WRITE_UINT32 (dst, t0);
    LE_WRITE_UINT32 (dst + 4, t1);
    LE_WRITE_UINT32 (dst + 8, t2);
    LE_WRITE_UINT32 (dst + 12, t3);
  }
}
```

Figure 9.2: Decryption Function

- Macros : They have been instantiated to perform the algorithm through SW implementation (RV32E) or HW implementation (RV32E+zknd+zkne). The Figure 9.3 shows how the macros has been implemented.

  – SW implementation: performs the layers of a round directly through precomputed look-up tables (T-Tables) with 256 entries of 32 bits (Appendix D) for the middle rounds and one precomputed table with 256 entries of 8 bits for the final round (Sbox of Appendix D).

– HW implementation: uses the zknd and zkne instructions, specified in chapter 6. The correct instruction is run 4 times for each call of the macro. So to run a full round, the instruction is performed 16 times.

```
#define ACCEL

#ifndef ACCEL

#define AES_ROUND(T, w0, w1, w2, w3, k)         \
((  T->table[0][ B0(w0) ]                        \
 ^ T->table[1][ B1(w1) ]                        \
 ^ T->table[2][ B2(w2) ]                        \
 ^ T->table[3][ B3(w3) ]) ^ (k))

#define AES_FINAL_ROUND(T, w0, w1, w2, w3, k)           \
 ((   (uint32_t) T->sbox[ B0(w0) ]                       \
  | ((uint32_t) T->sbox[ B1(w1) ] << 8)                  \
  | ((uint32_t) T->sbox[ B2(w2) ] << 16)                 \
  | ((uint32_t) T->sbox[ B3(w3) ] << 24)) ^ (k))

#define AES_ROUND_DEC(T, w0, w1, w2, w3, k)         \
((  T->table[0][ B0(w0) ]                        \
 ^ T->table[1][ B1(w1) ]                        \
 ^ T->table[2][ B2(w2) ]                        \
 ^ T->table[3][ B3(w3) ]) ^ (k))

#define AES_FINAL_ROUND_DEC(T, w0, w1, w2, w3, k)           \
 ((   (uint32_t) T->sbox[ B0(w0) ]                       \
  | ((uint32_t) T->sbox[ B1(w1) ] << 8)                  \
  | ((uint32_t) T->sbox[ B2(w2) ] << 16)                 \
  | ((uint32_t) T->sbox[ B3(w3) ] << 24)) ^ (k))

#else

#define AES_ROUND(T, w0, w1, w2, w3, k) \
__builtin_riscv_aes32esmi(__builtin_riscv_aes32esmi(__builtin_riscv_aes32esmi(__builtin_riscv_aes32esmi(k,w0,0),w1,1),w2,2),w3,3)

#define  AES_FINAL_ROUND(T, w0, w1, w2, w3, k) \
__builtin_riscv_aes32esi(__builtin_riscv_aes32esi(__builtin_riscv_aes32esi(__builtin_riscv_aes32esi(k,w0,0),w1,1),w2,2),w3,3)

#define AES_ROUND_DEC(T, w0, w1, w2, w3, k) \
__builtin_riscv_aes32dsmi(__builtin_riscv_aes32dsmi(__builtin_riscv_aes32dsmi(__builtin_riscv_aes32dsmi(k,w0,0),w1,1),w2,2),w3,3)

#define AES_FINAL_ROUND_DEC(T, w0, w1, w2, w3, k) \
__builtin_riscv_aes32dsi(__builtin_riscv_aes32dsi(__builtin_riscv_aes32dsi(__builtin_riscv_aes32dsi(k,w0,0),w1,1),w2,2),w3,3)

#endif
```

Figure 9.3: Macros for AES algorithm

- ACCEL flag: This flag is used, as the Figure 9.3 shows, to choose between the SW implementation or the HW implementation.

Results

The code has been organized to be compiled by the STxP5 Compiler, enabling it to run directly on the processor. Subsequently, the code was executed twice: first with the software (SW) and then with the hardware (HW) implementation. The assembly code regarding the loop of the middle rounds and the final round, without the assembly of full function, are shown in Figure 9.4 and in Figure 9.5 to make comparisons. The results obtained are shown in Table 9.1.

Software	• T-Tables: ∼8 kBytes of memory to be stored • RV32E instructions
Hardware	• Reduced code size (0.5x than T-tables) • No T-Tables: 0.0x Data Memory • 4x faster than T-tables in full function • 5x faster than T-tables in middle and final rounds • +1.1 kGates in ALU Area

Table 9.1: Comparison of SW and HW Implementations for AES-128

```
40225c: 23 26 d1 02   sw      a3, 44(sp)
402260: 93 82 07 00   mv      t0, a5
402264: 93 06 06 00   mv      a3, a2
402268: 13 04 05 00   mv      s0, a0
40226c: 13 f6 f7 0f   andi    a2, a5, 255
402270: 13 16 26 00   slli    a2, a2, 2
402274: 33 86 c4 00   add     a2, s1, a2
402278: 03 26 06 00   lw      a2, 0(a2)
  .
  .
4023fc: b3 c2 82 00   xor     t0, t0, s0
402400: 83 23 01 04   lw      t2, 64(sp)
402404: b3 c7 57 00   xor     a5, a5, t0
402408: 33 c3 e7 00   xor     t1, a5, a4
40240c: 83 27 81 03   lw      a5, 56(sp)
402410: 93 86 f6 ff   addi    a3, a3, -1
402414: 93 85 05 01   addi    a1, a1, 16
402418: e3 92 06 e4   bnez    a3, 0x40225c <_nettle_aes_encrypt+0x388>


402080: 03 27 81 03   lw      a4, 56(sp)
402084: 13 17 07 01   slli    a4, a4, 16
402088: 13 57 87 01   srli    a4, a4, 24
40208c: 33 07 e4 00   add     a4, s0, a4
402090: 03 47 07 00   lbu     a4, 0(a4)
402094: 93 f7 f5 0f   andi    a5, a1, 255
402098: b3 07 f4 00   add     a5, s0, a5
40209c: 83 c7 07 00   lbu     a5, 0(a5)
4020a0: 13 17 87 00   slli    a4, a4, 8
4020a4: 13 16 86 00   slli    a2, a2, 8
4020a8: 13 56 86 01   srli    a2, a2, 24
4020ac: 33 06 c4 00   add     a2, s0, a2
4020b0: 03 46 06 00   lbu     a2, 0(a2)
4020b4: 13 55 85 01   srli    a0, a0, 24
4020b8: 33 05 a4 00   add     a0, s0, a0
4020bc: 03 45 05 00   lbu     a0, 0(a0)
4020c0: 33 67 f7 00   or      a4, a4, a5
4020c4: 13 16 06 01   slli    a2, a2, 16
4020c8: 83 a6 c6 00   lw      a3, 12(a3)
4020cc: 13 15 85 01   slli    a0, a0, 24
4020d0: 33 65 a6 00   or      a0, a2, a0
4020d4: 33 65 a7 00   or      a0, a4, a0
4020d8: 33 45 d5 00   xor     a0, a0, a3
```

x4

Figure 9.4: Assembly code of SW implementation

```
40084e: 03 a5 43 ff   lw      a0, -12(t2)
400852: 3a 86         mv      a2, a4
400854: 33 05 15 26   aes32esmi       a0, a0, ra, 0
400858: 03 a3 83 ff   lw      t1, -8(t2)
40085c: 33 05 85 66   aes32esmi       a0, a0, s0, 1
400860: b3 06 e5 a6   aes32esmi       a3, a0, a4, 2
400864: 03 a5 c3 ff   lw      a0, -4(t2)
400868: b3 05 83 26   aes32esmi       a1, t1, s0, 0
40086c: b3 85 e5 66   aes32esmi       a1, a1, a4, 1
400870: b3 85 95 a6   aes32esmi       a1, a1, s1, 2
400874: 33 05 e5 26   aes32esmi       a0, a0, a4, 0
400878: 03 a3 03 00   lw      t1, 0(t2)
40087c: 33 05 95 66   aes32esmi       a0, a0, s1, 1
400880: 33 05 15 a6   aes32esmi       a0, a0, ra, 2
400884: 33 07 85 e6   aes32esmi       a4, a0, s0, 3
400888: 33 05 93 26   aes32esmi       a0, t1, s1, 0
40088c: 33 05 15 66   aes32esmi       a0, a0, ra, 1
400890: 33 05 85 a6   aes32esmi       a0, a0, s0, 2
400894: 33 84 15 e6   aes32esmi       s0, a1, ra, 3
400898: b3 80 96 e6   aes32esmi       ra, a3, s1, 3
40089c: b3 04 c5 e6   aes32esmi       s1, a0, a2, 3
4008a0: fd 12         addi    t0, t0, -1
4008a2: c1 03         addi    t2, t2, 16
4008a4: e3 95 02 fa   bnez    t0, 0x40084e <_nettle_aes_encrypt+0x196>


4006dc: 03 a5 02 00   lw      a0, 0(t0)
4006e0: 33 05 15 22   aes32esi        a0, a0, ra, 0
4006e4: 83 a5 42 00   lw      a1, 4(t0)
4006e8: 33 05 85 62   aes32esi        a0, a0, s0, 1
4006ec: 33 05 e5 a2   aes32esi        a0, a0, a4, 2
4006f0: 33 05 95 e2   aes32esi        a0, a0, s1, 3
4006f4: b3 85 85 22   aes32esi        a1, a1, s0, 0
4006f8: 03 a6 82 00   lw      a2, 8(t0)
4006fc: b3 85 e5 62   aes32esi        a1, a1, a4, 1
400700: b3 85 95 a2   aes32esi        a1, a1, s1, 2
400704: b3 85 15 e2   aes32esi        a1, a1, ra, 3
400708: 33 06 e6 22   aes32esi        a2, a2, a4, 0
40070c: 83 a2 c2 00   lw      t0, 12(t0)
400710: 33 06 96 62   aes32esi        a2, a2, s1, 1
400714: 33 06 16 a2   aes32esi        a2, a2, ra, 2
400718: 33 03 86 e2   aes32esi        t1, a2, s0, 3
40071c: 33 86 92 22   aes32esi        a2, t0, s1, 0
400720: 33 06 16 62   aes32esi        a2, a2, ra, 1
400724: 33 06 86 a2   aes32esi        a2, a2, s0, 2
400728: 33 06 e6 e2   aes32esi        a2, a2, a4, 3
```

Figure 9.5: Assembly code of HW implementation

9.2 SHA-256 algorithm

As for the AES-128, the SHA-256 algorithm has been run starting form a specific benchmark taken from Embech open source benchmark suite [29]. The code is based on 3 main parts:

- 1 main function:
  - `_nettle_sha256_compress`: Function that performs the calculations of the algorithm shown in Appendix C. Figure 9.6 shows how it works.

```
void
_nettle_sha256_compress (uint32_t * state, const uint8_t * input,
                         const uint32_t * k)
{
  uint32_t data[SHA256_DATA_LENGTH];
  uint32_t A, B, C, D, E, F, G, H;      /* Local vars */
  unsigned i;
  //unsigned j;
  //uint32_t j;
  uint32_t *d;

  for (i = 0; i < SHA256_DATA_LENGTH; i++, input += 4)
    {
      data[i] = READ_UINT32 (input);
    }

  /* Set up first buffer and local data buffer */
  A = state[0];
  B = state[1];
  C = state[2];
  D = state[3];
  E = state[4];
  F = state[5];
  G = state[6];
  H = state[7];

  /* Heavy mangling */
  /* First 16 subrounds that act on the original data */

  DEBUG (-1);
  for (i = 0, d = data; i < 16; i += 8, k += 8, d += 8)
    {
      ROUND (A, B, C, D, E, F, G, H, k[0], d[0]);
      DEBUG (i);
      ROUND (H, A, B, C, D, E, F, G, k[1], d[1]);
      DEBUG (i + 1);
      ROUND (G, H, A, B, C, D, E, F, k[2], d[2]);
      ROUND (F, G, H, A, B, C, D, E, k[3], d[3]);
      ROUND (E, F, G, H, A, B, C, D, k[4], d[4]);
      ROUND (D, E, F, G, H, A, B, C, k[5], d[5]);
      ROUND (C, D, E, F, G, H, A, B, k[6], d[6]);
      DEBUG (i + 6);
      ROUND (B, C, D, E, F, G, H, A, k[7], d[7]);
      DEBUG (i + 7);
    }

  for (i = 16; i < 64; i += 16, k += 16)
    {
      ROUND (A, B, C, D, E, F, G, H, k[0], EXPAND (data, 0));
      DEBUG (i);
      ROUND (H, A, B, C, D, E, F, G, k[1], EXPAND (data, 1));
      DEBUG (i + 1);
      ROUND (G, H, A, B, C, D, E, F, k[2], EXPAND (data, 2));
      DEBUG (i + 2);
      ROUND (F, G, H, A, B, C, D, E, k[3], EXPAND (data, 3));
      DEBUG (i + 3);
      ROUND (E, F, G, H, A, B, C, D, k[4], EXPAND (data, 4));
      DEBUG (i + 4);
      ROUND (D, E, F, G, H, A, B, C, k[5], EXPAND (data, 5));
      DEBUG (i + 5);
      ROUND (C, D, E, F, G, H, A, B, k[6], EXPAND (data, 6));
      DEBUG (i + 6);
      ROUND (B, C, D, E, F, G, H, A, k[7], EXPAND (data, 7));
      DEBUG (i + 7);
      ROUND (A, B, C, D, E, F, G, H, k[8], EXPAND (data, 8));
      DEBUG (i + 8);
      ROUND (H, A, B, C, D, E, F, G, k[9], EXPAND (data, 9));
      DEBUG (i + 9);
      ROUND (G, H, A, B, C, D, E, F, k[10], EXPAND (data, 10));
      DEBUG (i + 10);
      ROUND (F, G, H, A, B, C, D, E, k[11], EXPAND (data, 11));
      DEBUG (i + 11);
      ROUND (E, F, G, H, A, B, C, D, k[12], EXPAND (data, 12));
      DEBUG (i + 12);
      ROUND (D, E, F, G, H, A, B, C, k[13], EXPAND (data, 13));
      DEBUG (i + 13);
      ROUND (C, D, E, F, G, H, A, B, k[14], EXPAND (data, 14));
      DEBUG (i + 14);
      ROUND (B, C, D, E, F, G, H, A, k[15], EXPAND (data, 15));
      DEBUG (i + 15);

    }

  /* Update state */
  state[0] += A;
  state[1] += B;
  state[2] += C;
  state[3] += D;
  state[4] += E;
  state[5] += F;
  state[6] += G;
  state[7] += H;
}
```

Figure 9.6: SHA-256 main function

- Macros : They have been instantiated to perform the algorithm through SW implementation (RV32E) or HW implementation (RV32E+zknh). The Figure 9.7 shows how the macros has been implemented.

  – SW implementation: performs the Sigma0, Sigma1, Sum0 and

Sum1 functions directly with the formulas written in the specifications of the algorithm [10].

– HW implementation: performs the Sigma0, Sigma1, Sum0 and Sum1 functions through the instructions of zknh extension specified in chapter 6.

```
#define SHA256_DATA_LENGTH 16

#define DEBUG(i)

#define ACCEL

#ifndef ACCEL
#define Choice(x,y,z)   ( (z) ^ ( (x) & ( (y) ^ (z) ) ) )
#define Majority(x,y,z) ( ((x) & (y)) ^ ((z) & ((x) ^ (y))) )

#define S0(x) (ROTL32(30,(x)) ^ ROTL32(19,(x)) ^ ROTL32(10,(x)))
#define S1(x) (ROTL32(26,(x)) ^ ROTL32(21,(x)) ^ ROTL32(7,(x)))

#define s0(x) (ROTL32(25,(x)) ^ ROTL32(14,(x)) ^ ((x) >> 3))
#define s1(x) (ROTL32(15,(x)) ^ ROTL32(13,(x)) ^ ((x) >> 10))

#else

#define Choice(x,y,z)   ( (z) ^ ( (x) & ( (y) ^ (z) ) ) )
#define Majority(x,y,z) ( ((x) & (y)) ^ ((z) & ((x) ^ (y))) )


#define S0(x) __builtin_riscv_sha256sum0(x)
#define S1(x) __builtin_riscv_sha256sum1(x)


#define s0(x) __builtin_riscv_sha256sig0(x)
#define s1(x) __builtin_riscv_sha256sig1(x)

#endif

#define EXPAND(W,i) \
( W[(i) & 15 ] += (s1(W[((i)-2) & 15]) + W[((i)-7) & 15] + s0(W[((i)-15) & 15])) )

/* It's crucial that DATA is only used once, as that argument will
 * have side effects. */
#define ROUND(a,b,c,d,e,f,g,h,k,data) do {      \
    h += S1(e) + Choice(e,f,g) + k + data;      \
    d += h;                                     \
    h += S0(a) + Majority(a,b,c);               \
  } while (0)
```

Figure 9.7: Macros for SHA algorithm

- ACCEL flag: This flag is used, as the Figure 9.7 shows, to choose between the SW implementation or the HW implementation.

Results

The code has been organized to be compiled by the STxP5 Compiler, enabling it to run directly on the processor. Subsequently, the code was executed twice: first with the software (SW) implementation, the

assembly code of a function is shown in Figure 9.9, and then with the hardware (HW) implementation, the assembly code of a function is shown in Figure 9.8, to makes some comparisons. The results obtained are shown in Table 9.2.

Software	• RV32E: the formulas have to be performed with simple instructions.
Hardware	• Reduced code size (0.5x than SW) • 2x faster than T-tables • +1.0 kGates in ALU Area

Table 9.2: Comparison of SW and HW Implementations for SHA-256

```
409e88: 23 2e 91 04   sw      s1, 92(sp)
409e8c: 23 20 f1 08   sw      a5, 128(sp)
409e90: 93 90 14 10   sha256sum1      ra, s1
```

Figure 9.8: Assembly of a function in SHA-256 Algorithm (HW)

```
409e8c: 23 26 a1 06   sw      a0, 108(sp)
409e90: 93 d0 67 00   srli    ra, a5, 6
409e94: 13 95 a7 01   slli    a0, a5, 26
409e98: 33 65 15 00   or      a0, a0, ra
409e9c: 93 d0 b7 00   srli    ra, a5, 11
409ea0: 23 20 f1 08   sw      a5, 128(sp)
409ea4: 13 97 57 01   slli    a4, a5, 21
409ea8: 33 67 17 00   or      a4, a4, ra
409eac: 33 45 e5 00   xor     a0, a0, a4
409eb0: 13 d7 97 01   srli    a4, a5, 25
409eb4: 93 90 77 00   slli    ra, a5, 7
409eb8: 33 e7 e0 00   or      a4, ra, a4
409ebc: 33 45 e5 00   xor     a0, a0, a4
409ec0: 83 24 41 08   lw      s1, 132(sp)
409ec4: 83 22 41 07   lw      t0, 116(sp)
409ec8: 33 c7 92 00   xor     a4, t0, s1
409ecc: 33 77 f7 00   and     a4, a4, a5
409ed0: 83 20 06 00   lw      ra, 0(a2)
409ed4: 23 2c d1 06   sw      a3, 120(sp)
409ed8: 83 a6 06 00   lw      a3, 0(a3)
409edc: 33 47 57 00   xor     a4, a4, t0
```

Figure 9.9: Assembly of a function in SHA-256 Algorithm (SW)

Chapter 10

Conclusions and Future Works

To sum up the thesis work, the Scalar Cryptography Extension has been implemented to bring some improvements to specific algorithms while minimizing the impact of the extensions on the processor's area. Table 10.1 summarizes the results obtained. The results are consistent with the expected outcomes specified in the Background chapter 2.2. Overall, with all the extensions implemented, there is an increase of +2.4 kG in the ALU area. The decoder and controller are not considered because there is not an important impact on them. So since the full implementation of the processors is 38.7 kG, the total increase of area due to the extensions is equal to the 6% respect to the total area. In this scenario, several future works that can be undertaken. Firstly, benchmarking on SHA-512 to evaluate the performance of the remaining part of the extension zknh that has not been performed. Secondly, each extension could be run with a benchmark to evaluate the type of improvement It can bring to the processor. Thirdly, there could be other optimizations to be done both on the instructions side and on the compiler side.

Extension	**Instructions**	**Alu Area**	**Benchmark**
Baseline	RV32E	3.1 kGates	• RV32E instructions • T-Tables for AES-128 algorithm
Zbkb	+5	+0.1 kG	
Zbkx	+2	+0.5 kG	
Zknh	+10	+1.0 kG	For SHA-256: • 0.5x code size • 2x faster
Zknd	+2	+0.7 kG	
Zkne	+2	+0.7 kG	
Zknd+Zkne	+4	+1.1 kG	For AES-128: • 0.5x code size • 4x faster • 5x faster than T-tables in middle and final rounds • 0.0x Data Memory
All Crypto	+21	+2.4 kG	

Table 10.1: Results

Appendix A

nML & pdg implementation

An example of how instructions are implemented with nML and pdg is shown below:

nML

```
enum funct10_rrr_zbkx{
    xperm8 = 0b0010100100,
    xperm4 = 0b0010100010
};

opn zbkx_rrr_instr (op: funct10_rrr_zbkx, rd: eX, rs1: eX,
   rs2: eX)
{
    LLVM( isa = base; class = compute; )
    action {
    stage DOF:
      ill_opc_DOF   = 0;
      pEX_op1_q = DOF_op1 = r1 = X[rs1];
      pEX_op2_q = DOF_op2 = r2 = X[rs2];

      OCD_DOF(isa32)
    stage EX:
```

```
17        aluA = EX_dp_in1 = pEX_op1_q;
18        aluB = EX_dp_in2 = pEX_op2_q;
19
20        switch(op) {
21 #if defined(__programmers_view__ )
22 #else
23 #endif
24
25      case xperm8 : aluR = xperm8(aluA,aluB) @alu;
26      case xperm4 : aluR = xperm4(aluA,aluB) @alu;
27        }
28          pWB_op1_q = EX_op1 = byp_EX = aluR;
29
30          OCD_EX(EX_dp_in1, EX_dp_in2)
31
32        stage WB:
33
34          if (rd: x0)           aluR_dead = pWB_op1_q;
35          else                  X[rd] = w1 = pWB_op1_q;
36
37          OCD_WB()
38
39      }
40      syntax :   op   PADMNM " " rd "," PADOP1 rs1 "," PADOP2
    rs2 ;
41
42      image :   op[9..3]::RVIE(rs2)::RVIE(rs1)::op[2..0]::RVIE(
    rd) ;
43 }
```

pdg

```
1 class ubyte property( vector uint8_t[4] );
2 class vbyte property( vector uint4_t[8] );
3
4 w32 xperm8 (w32 a, w32 b){
5     uint32_t r = 0;
6     ubyte temp1;
7     ubyte temp2;
```

```
8       temp1[0] =  a[ 7: 0] ;
9       temp1[1] =  a[15: 8] ;
10      temp1[2] =  a[23:16] ;
11      temp1[3] =  a[31:24] ;
12
13      temp2[0] =  b[ 7: 0] ;
14      temp2[1] =  b[15: 8] ;
15      temp2[2] =  b[23:16] ;
16      temp2[3] =  b[31:24] ;
17
18      uint8_t si = 0;
19
20      for (int i = 0;i<4;i++){
21          switch(temp2[i]){
22              case 0:  si = temp1[0];break;
23              case 1:  si = temp1[1];break;
24              case 2:  si = temp1[2];break;
25              case 3:  si = temp1[3];break;
26              default: si = 0;break;
27          }
28          r |= (uint32_t)si << (i << 3);
29      }
30      return r;
31  }
32
33  w32 xperm4 (w32 a, w32 b){
34      uint32_t r = 0;
35      vbyte temp1;
36      vbyte temp2;
37      temp1[0] =  a[ 3: 0] ;
38      temp1[1] =  a[7: 4] ;
39      temp1[2] =  a[11:8] ;
40      temp1[3] =  a[15:12] ;
41      temp1[4] =  a[19:16] ;
42      temp1[5] =  a[23:20] ;
43      temp1[6] =  a[27:24] ;
44      temp1[7] =  a[31:28] ;
45
46      temp2[0] =  b[ 3: 0] ;
47      temp2[1] =  b[7: 4] ;
48      temp2[2] =  b[11:8] ;
49      temp2[3] =  b[15:12] ;
```

```
50      temp2[4] =  b[19:16] ;
51      temp2[5] =  b[23:20] ;
52      temp2[6] =  b[27:24] ;
53      temp2[7] =  b[31:28] ;
54
55      uint4_t si = 0;
56
57      for (int i = 0;i<8;i++){
58          switch(temp2[i]){
59              case 0:  si = temp1[0];break;
60              case 1:  si = temp1[1];break;
61              case 2:  si = temp1[2];break;
62              case 3:  si = temp1[3];break;
63              case 4:  si = temp1[4];break;
64              case 5:  si = temp1[5];break;
65              case 6:  si = temp1[6];break;
66              case 7:  si = temp1[7];break;
67              default: si = 0;break;
68          }
69          r |= (uint32_t)si << (i << 2);
70      }
71      return r;
72  }
```

Appendix B

S-Box

The Substitution Box (S-Box) is one of the main components, designed to perform substitution operations, of symmetric key algorithms as for example AES algorithm. In the case of block ciphers, S-Boxes are used to obscure the relationship between the encryption key and the resulting ciphertext, ensuring confusion. S-Box function is a non-linear transformation, It can be implemented trough a $m \times n$ S-Box as a lookup table with 2^m words, each containing n bits. So It replaces an m-bit input with a specific n-bit output based on a predefined substitution table. The sizes of the input (m) and output (n) can be different. In the case of Zbkx, the Crossbar Permutation involves rearranging or shuffling the bits of data in a key-dependent manner. This permutation technique is crucial for introducing confusion and non-linearity into S-Boxes.

Appendix C

SHA2

Algorithm

The SHA-256 and SHA-512 algorithms divide the input message into 512-bit and 1024-bit input blocks, respectively. Then, they apply the round function, shown in Figure C.1, to each block. The SHA-256 and SHA-512 algorithms use 64 and 80 rounds, respectively. The SHA-512 hash function computation is identical to that of the SHA-256 hash function, with the following differences:

- The size of the operands: 64 bits for SHA-512 instead of 32 bits for SHA-256.

- The size of the Digest Message: 512 bits for SHA-512, 256 bits for the SHA-256.

- The Σ, σ functions.

- The values Wt and Kt: 64 bits for SHA-512 instead of 32 bits for SHA-256.

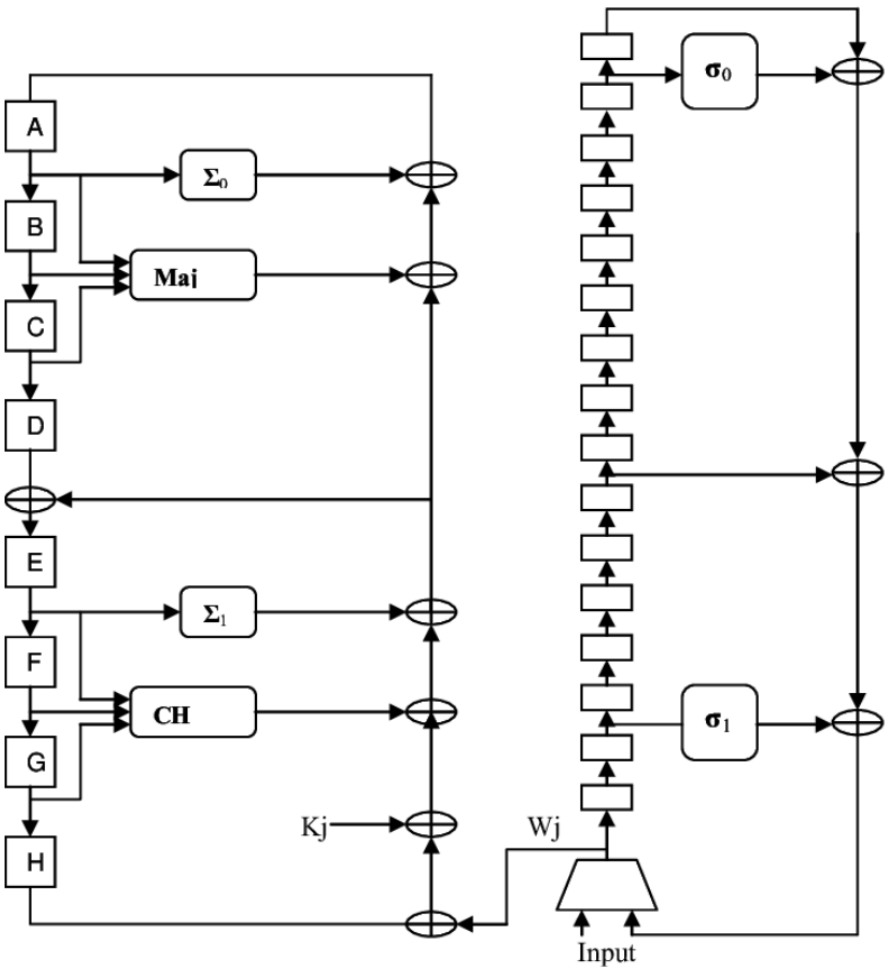

Figure C.1: SHA-256 and SHA-512 structure [30]

Appendix D

AES

Key Schedule

The AES key schedule takes the primary key and, for each round of AES, generates round key (k). The number of round subkeys is equal to the number of rounds plus one. Therefore, for a key length of 128 bits, the number of rounds is nr = 10, resulting in 11 subkeys, each of 128 bits. For AES with a 192-bit key and so 12 rounds, it requires 13 subkeys of length 128 bits, and AES with a 256-bit key (14 rounds) requires 15 subkeys. Each round key has the same size as the primary key. The function G uses an 8-bit round constant (rc) for each round. The key schedule for AES-128, along with the g function, is illustrated in Figure D.1. The subkeys are stored in a key expansion array with elements W[0],...,W[43]. The elements K0,...,K15 represent the bytes of the original AES key. The first subkey word is computed using the G function, while the other three words are calculated recursively. The function G function is a nonlinear function with a four-byte input and output in which the four input bytes are rotated, a byte-wise S-Box substitution is performed, and then a round coefficient RC is added to them. The round coefficients differ from round to round [24].

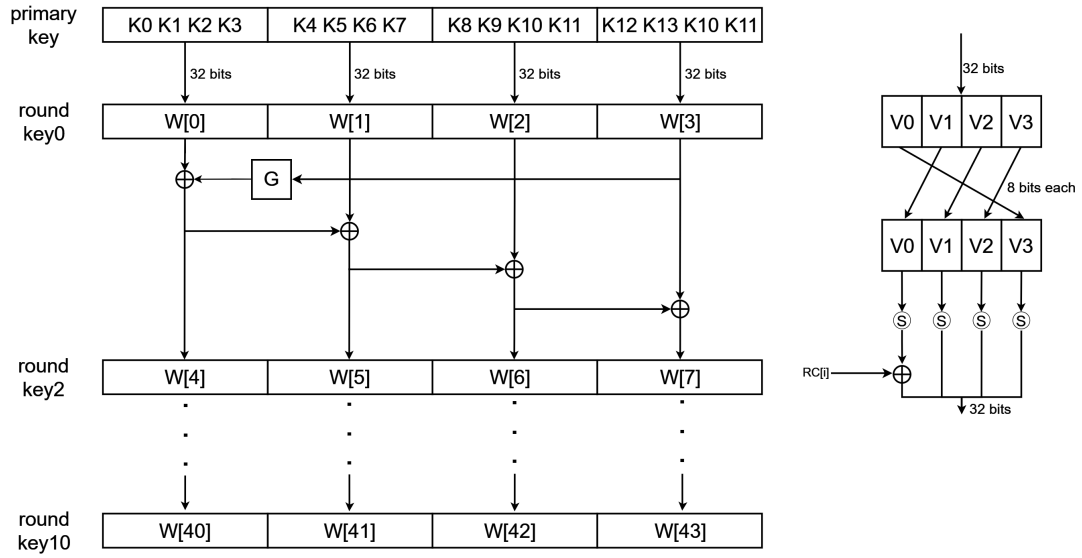

Figure D.1: Key Schedule for AES-128

Forward Sbox - SubBytes

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure D.2: Fwd Sbox - Precalculated SubByte [31]

Inverse SBox - SubBytes

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
10	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
20	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
30	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
40	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
50	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
60	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
70	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
80	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
90	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a0	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b0	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c0	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d0	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e0	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f0	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figure D.3: Inv Sbox - Precalculated SubByte Inversion [31]

T-tables middle round

There are 8 different T-tables used in the code (4 for Encryption and 4 for Decryption), Figure D.4 shows an example.

```
{
 0x50a7f451, 0x5365417e, 0xc3a4171a, 0x965e273a,
 0xcb6bab3b, 0xf1459d1f, 0xab58faac, 0x9303e34b,
 0x55fa3020, 0xf66d76ad, 0x9176cc88, 0x254c02f5,
 0xfcd7e54f, 0xd7cb2ac5, 0x80443526, 0x8fa362b5,
 0x495ab1de, 0x671bba25, 0x980eea45, 0xe1c0fe5d,
 0x02752fc3, 0x12f04c81, 0xa397468d, 0xc6f9d36b,
 0xe75f8f03, 0x959c9215, 0xeb7a6dbf, 0xda595295,
 0x2d83bed4, 0xd3217458, 0x2969e049, 0x44c8c98e,
 0x6a89c275, 0x78798ef4, 0x6b3e5899, 0xdd71b927,
 0xb64fe1be, 0x17ad88f0, 0x66ac20c9, 0xb43ace7d,
 0x184adf63, 0x82311ae5, 0x60335197, 0x457f5362,
 0xe07764b1, 0x84ae6bbb, 0x1ca081fe, 0x942b08f9,
 0x58684870, 0x19fd458f, 0x876cde94, 0xb7f87b52,
 0x23d373ab, 0xe2024b72, 0x578f1fe3, 0x2aab5566,
 0x0728ebb2, 0x03c2b52f, 0x9a7bc586, 0xa50837d3,
 0xf2872830, 0xb2a5bf23, 0xba6a0302, 0x5c8216ed,
 0x2b1ccf8a, 0x92b479a7, 0xf0f207f3, 0xa1e2694e,
 0xcdf4da65, 0xd5be0506, 0x1f6234d1, 0x8afea6c4,
 0x9d532e34, 0xa055f3a2, 0x32e18a05, 0x75ebf6a4,
 0x39ec830b, 0xaaef6040, 0x069f715e, 0x51106ebd,
 0xf98a213e, 0x3d06dd96, 0xae053edd, 0x46bde64d,
 0xb58d5491, 0x055dc471, 0x6fd40604, 0xff155060,
 0x24fb9819, 0x97e9bdd6, 0xcc434089, 0x779ed967,
 0xbd42e8b0, 0x888b8907, 0x385b19e7, 0xdbeec879,
 0x470a7ca1, 0xe90f427c, 0xc91e84f8, 0x00000000,
 0x83868009, 0x48ed2b32, 0xac70111e, 0x4e725a6c,
 0xfbff0efd, 0x5638850f, 0x1ed5ae3d, 0x27392d36,
 0x64d90f0a, 0x21a65c68, 0xd1545b9b, 0x3a2e3624,
 0xb1670a0c, 0x0fe75793, 0xd296eeb4, 0x9e919b1b,
 0x4fc5c080, 0xa220dc61, 0x694b775a, 0x161a121c,
 0x0aba93e2, 0xe52aa0c0, 0x43e0223c, 0x1d171b12,
 0x0b0d090e, 0xadc78bf2, 0xb9a8b62d, 0xc8a91e14,
 0x8519f157, 0x4c0775af, 0xbbdd99ee, 0xfd607fa3,
 0x9f2601f7, 0xbcf5725c, 0xc53b6644, 0x347efb5b,
 0x7629438b, 0xdcc623cb, 0x68fcedb6, 0x63f1e4b8,
 0xcadc31d7, 0x10856342, 0x40229713, 0x2011c684,
 0x7d244a85, 0xf83dbbd2, 0x1132f9ae, 0x6da129c7,
 0x4b2f9e1d, 0xf330b2dc, 0xec52860d, 0xd0e3c177,
 0x6c16b32b, 0x99b970a9, 0xfa489411, 0x2264e947,
 0xc48cfca8, 0x1a3ff0a0, 0xd82c7d56, 0xef903322,
 0xc74e4987, 0xc1d138d9, 0xfea2ca8c, 0x360bd498,
 0xcf81f5a6, 0x28de7aa5, 0x268eb7da, 0xa4bfad3f,
 0xe49d3a2c, 0x0d927850, 0x9bcc5f6a, 0x62467e54,
 0xc2138df6, 0xe8b8d890, 0x5ef7392e, 0xf5afc382,
 0xbe805d9f, 0x7c93d069, 0xa92dd56f, 0xb31225cf,
 0x3b99acc8, 0xa77d1810, 0x6e639ce8, 0x7bbb3bdb,
 0x097826cd, 0xf418596e, 0x01b79aec, 0xa89a4f83,
 0x656e95e6, 0x7ee6ffaa, 0x08cfbc21, 0xe6e815ef,
 0xd99be7ba, 0xce366f4a, 0xd4099fea, 0xd67cb029,
 0xafb2a431, 0x31233f2a, 0x3094a5c6, 0xc066a235,
 0x37bc4e74, 0xa6ca82fc, 0xb0d090e0, 0x15d8a733,
 0x4a9804f1, 0xf7daec41, 0x0e50cd7f, 0x2ff69117,
 0x8dd64d76, 0x4db0ef43, 0x544daacc, 0xdf0496e4,
 0xe3b5d19e, 0x1b886a4c, 0xb81f2cc1, 0x7f516546,
 0x04ea5e9d, 0x5d358c01, 0x737487fa, 0x2e410bfb,
 0x5a1d67b3, 0x52d2db92, 0x335610e9, 0x1347d66d,
 0x8c61d79a, 0x7a0ca137, 0x8e14f859, 0x893c13eb,
 0xee27a9ce, 0x35c961b7, 0xede51ce1, 0x3cb1477a,
 0x59dfd29c, 0x3f73f255, 0x79ce1418, 0xbf37c773,
 0xeacdf753, 0x5baafd5f, 0x146f3ddf, 0x86db4478,
 0x81f3afca, 0x3ec468b9, 0x2c342438, 0x5f40a3c2,
 0x72c31d16, 0x0c25e2bc, 0x8b493c28, 0x41950dff,
 0x7101a839, 0xdeb30c08, 0x9ce4b4d8, 0x90c15664,
 0x6184cb7b, 0x70b632d5, 0x745c6c48, 0x4257b8d0,
 },
```

Figure D.4: Precalculated T-Table

Bibliography

[1] IEEE Spectrum. *Chip Hall of Fame: Motorola MC68000 Microprocessor*. URL: https://spectrum.ieee.org/chip-hall-of-fame-motorola-mc68000-microprocessor (cit. on p. 1).

[2] Jaikrishnan Menon Emily Blem and Karthikeyan Sankaralingam. «Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures». In: *19th IEEE International Symposium on High Performance Computer Architecture (HPCA 2013)*. University of Wisconsin - Madison, 2013 (cit. on p. 1).

[3] IBM. *Reduced instruction set computer (RISC) architecture*. URL: https://www.ibm.com/history/risc (cit. on p. 2).

[4] Shreyas Sharma. *RISC-V vs ARM: A Comprehensive Comparison of Processor Architectures*. Aug. 2023. URL: https://www.wevolver.com/article/risc-v-vs-arm (cit. on p. 2).

[5] *History of RISC-V*. URL: https://riscv.org/about/history/ (cit. on p. 2).

[6] *RISC-V Specifications*. URL: https://github.com/riscv?q=&type=all&language=&sort= (cit. on p. 2).

[7] RISC-V International. «RISC-V Cryptography Extensions Volume I: Scalar & Entropy Source Instructions». In: Feb. 2022 (cit. on pp. 2, 3, 33–38, 40, 41).

[8] RISC-V International. «RISC-V Cryptography Extensions Volume II: Vector Instructions». In: Oct. 2023 (cit. on p. 2).

[9] National Institute of Standards and Technology. «ADVANCED ENCRYPTION STANDARD (AES)». In: Nov. 2001 (cit. on p. 3).

[10] National Institute of Standards and Technology. «SECURE HASH STANDARD». In: Aug. 2015 (cit. on pp. 3, 35, 36, 93).

[11] David Patterson Andrew Waterman Yunsup Lee and Krste Asanovi´. «The RISC-V Instruction Set Manual Volume I: User-Level ISA». In: CS Division, EECS Department, University of California, Berkeley, May 2014 (cit. on pp. 5, 7, 8).

[12] Krste Asanovi´ Andrew Waterman. «The RISC-V Instruction Set Manual Volume II: Privileged Architecture». In: CS Division, EECS Department, University of California, Berkeley, May 2017 (cit. on pp. 5, 6, 18).

[13] Christof Paar Adam J. Elbirt. «An Instruction-Level Distributed Processor for Symmetric-Key Cryptography». In: 2005 (cit. on p. 11).

[14] Arul Paniandi Mohamed Khalil Hani Hau Yuan Wen. «DESIGN AND IMPLEMENTATION OF A PRIVATE AND PUBLIC KEY CRYPTO PROCESSOR FOR NEXT-GENERATION IT SECURITY APPLICATIONS». In: 2006 (cit. on p. 12).

[15] TAO LU. «A Survey on RISC-V Security: Hardware and Architecture». In: 9 Jul 2021 (cit. on p. 12).

[16] Ben Marshall G. Richard Newell Dan Page Markku-Juhani O. Saarinen Claire Wolf. «The design of scalar AES Instruction Set Extensions for RISC-V». In: 2020 (cit. on pp. 12, 68, 69).

[17] *Improvement SHA-256 instructions*. URL: `https://github.com/mjosaarinen/lwsha_isa/blob/master/README.md` (cit. on p. 12).

[18] *Area of AES and SHA*. URL: `https://github.com/riscv/riscv-crypto/blob/main/rtl/README.md` (cit. on p. 12).

[19] Synopsis. *ASIP Designer: Application-Specific Processor Design Made Easy*. URL: `https://www.synopsys.com/dw/doc.php/ds/cc/asip-brochure.pdf` (cit. on p. 25).

[20] Gert Goossens. *Under the Hood of ASIP Designer - Application-Specific Processor Design Made Possible by Tool Automation*. URL: https://www.synopsys.com/designware-ip/technical-bulletin/under-the-hood-asip-designer.html (cit. on p. 25).

[21] Synopsys. «ASIP Designer nML Manual: The nML Processor Description Language». In: Dec. 2023 (cit. on p. 26).

[22] Synopsys. «ASIP Designer PDG Manual: Primitives Definition and Generation». In: Dec. 2023 (cit. on p. 29).

[23] Richard Newell. *Scalar Cryptography Instruction Set Extension Group Names Diagram*. June 2021. URL: https://wiki.riscv.org/display/HOME/Scalar+Cryptography+Instruction+Set+Extension+Group+Names+Diagram (cit. on p. 32).

[24] Jan Pelzl Christof Paar. «Understanding Cryptography». In: 2010 (cit. on pp. 39, 57, 58, 62, 105).

[25] Renè Peralta Joan Boyar. «A Small Depth-16 Circuit for the AES S-Box». In: 2011 (cit. on pp. 57, 62).

[26] Shay Gueron. «Intel Advanced Encryption Standard (AES) New Instrucitons Set». In: Jan. 2010 (cit. on p. 68).

[27] Sean Gulley Vinodh Gopal Kirk Yap Wajdi Feghali Jim Guilford Gil Wolrich. «Intel SHA Extensions». In: 2013 (cit. on p. 69).

[28] Arm. «Arm Architecture Reference Manual». In: 2013-2024 (cit. on pp. 69, 70).

[29] *Embech open source benchmark suite*. URL: https://github.com/embench/embench-iot (cit. on pp. 85, 91).

[30] Medien Zeghid Belgacem Bouallegue Mohsen Machhout Adel Baganne Rached Tourki. «Architectural design features of a programmable high throughput reconfigurable SHA-2 Processor». In: 2007 (cit. on p. 104).

[31] Gregory Durgin Brian Degnan. «A reference implementation of the AES S-Box». In: 2020 (cit. on pp. 106, 107).