VIABILITY AND IMPLEMENTATION OF A VECTOR CRYPTOGRAPHY EXTENSION FOR RISC-V

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Jonathan Skelly

June 2022

COMMITTEE MEMBERSHIP

TITLE:	Viability and Implementation of a Vector Cryptography Extension for RISC-V
AUTHOR:	Jonathan Skelly
DATE SUBMITTED:	June 2022
COMMITTEE CHAIR:	Andrew Danowitz, Ph.D. Associate Professor of Computer Engineering
COMMITTEE MEMBER:	James Mealy, Ph.D. Professor of Computer Engineering
COMMITTEE MEMBER:	Joseph Callenes-Sloan, Ph.D. Associate Professor of Computer Engineering
COMMITTEE MEMBER:	Maria Pantoja, Ph.D. Associate Professor of Computer Engineering

ABSTRACT

Viability and Implementation of a Vector Cryptography Extension for RISC-V

Jonathan Skelly

RISC-V is an open-source instruction-set architecture (ISA) forming the basis of thousands of commercial and experimental microprocessors. The Scalar Cryptography extension ratified in December 2021 added scalar instructions that target common hashing and encryption algorithms, including SHA2 and AES. The next step forward for the RISC-V ISA in the field of cryptography and digital security is the development of vector cryptography instructions.

This thesis examines if it is viable to add vector implementations of existing RISC-V scalar cryptography instructions to the existing vector instruction format, and what improvements they can make to the execution of SHA2 and AES algorithms. Vector cryptography instructions vaeses, vaesesm, vaesds, vaesdsm, vsha256sch, and vsha256hash are proposed to optimize AES encryption and decryption, SHA256 message scheduling, and SHA256 hash rounds, with pseudocode, assembly examples, and a full 32-bit instruction format for each. Both algorithms stand to benefit greatly from vector instructions in reduction of computation time, code length, and instruction memory utilization due to large operand sizes and frequently repeated functions. As a proof of concept for the vector cryptography operations proposed, a full vector-based AES-128 encryption and SHA256 message schedule generation are performed on the 32-bit RISC-V Ibex processor and 128-bit Vicuna Vector Coprocessor in the Vivado simulation environment. Not counting stores or loads for fair comparison, the new Vector Cryptography extension completes a full encryption round in a single instruction compared to sixteen with the scalar extension, and can generate eight SHA256 message schedule double-words in a single instruction compared to the forty necessary on the scalar extension. These represent a 93.75% and 97.5% reduction in required instructions and memory for these functions respectively, at a hardware cost of 19.4% more LUTs and 1.44% more flip-flops on the edited Vicuna processor compared to the original.

Keywords: RISCV, Cryptography, Vector, ISA, Verilog, AES, SHA, Security, SIMD, Assembly

TABLE OF CONTENTS

Page

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER

1. INTRODUCTION 1

1.1 Statement of Problem 1

1.2 List of Terms 2

1.3 Purpose of Study 4

2. LITERATURE REVIEW 6

2.1 Advanced Encryption Standard (AES) 6

2.1.1 AES Key Schedule 7

2.1.2 Encryption 9

2.1.3 Decryption 11

2.2 Secure Hashing Algorithm 2 (SHA2) 14

2.2.1 Secure Hashing Algorithm 256 (SHA256) 14

2.3 RISC-V 19

2.3.1 RISC-V Vector Extension 20

2.3.2 RISC-V Scalar Cryptography Extension 21

2.3.2.1 RISC-V Scalar AES Instructions 22

2.3.2.1.1 AES32ESMI and AES32ESI 23

2.3.2.1.2 AES32DSMI and AES32DSI 25

2.3.2.2 RISC-V Scalar SHA2 Instructions 27

2.3.2.2.1 SHA256SIG0 and SHA256SIG1 27

2.3.2.2.2 SHA256SUM0 and SHA256SUM1 29

2.4 Existing Vector Cryptography Instructions 30

2.4.1 Intel AES Instructions 31

2.4.2 Intel SHA256 Instructions 32

2.5 Vicuna Vector Coprocessor 34
3. ANALYSIS 35
3.1 Methods 35
3.2 RISC-V Vector AES Instructions 36
3.2.1 VAESESM and VAESES 37
3.2.2 VAESDSM and VAESDS 38
3.3 RISC-V Vector SHA256 Instructions 40
3.3.1 VSHA256SCH 41
3.3.2 VSHA256HASH 42
3.4 Vivado Instruction Simulation on Vicuna 44
3.4.1 Full Vector AES Encryption Hardware Simulation 46
3.4.2 Full Vector SHA Message Schedule Hardware Simulation 50
4. CONCLUSION 55
REFERENCES 57
APPENDICES
A: Verilog Additions to Vicuna 59
B: Assembly Code 64

LIST OF TABLES

Table Page

1. AES Type Information 6
2. AES Key Expansion Information 7
3. AES Substitution Lookup Table 10
4. AES Inverse Substitution Lookup Table 13
5. SHA256 Round Constants 15
6. SHA256 Initial Hash Values 16
7. Formats for Vector Integer Arithmetic Instructions 20
8. Formats for 32-bit Scalar Cryptography AES Instructions 23
9. Formats for Scalar Cryptography SHA Instructions 27
10. Intel's Register Placement of SHA256 Message Schedule 33
11. Formats for Vector Cryptography AES Instructions 37
12. Formats for Vector Cryptography SHA256 Instructions 41
13. AES Example Input, Round Keys, and Outputs 47
14. AES Encryption Round Vector vs. Scalar 50
15. SHA256 Example Input, First 8 Double-Words, and Last 8 Double-Words 51
16. Generation of 8 SHA256 Message Schedule Double-Words Vector vs. Scalar 54

LIST OF FIGURES

Figure Page

1. AES Key Schedule Generation Workflow 8
2. AES Encryption Workflow 9
3. AES Decryption Workflow 12
4. SHA256 Message Decomposition and Padding 15
5. SHA256 Hashing Workflow 16
6. RISC-V AES Encrypt Instructions Assembly Syntax 23
7. SAIL Pseudocode for RISC-V AES Encrypt Instructions 24
8. RISC-V AES Encrypt Instructions Assembly Use Case 24
9. RISC-V AES Decrypt Instructions Assembly Syntax 25
10. SAIL Pseudocode for RISC-V AES Decrypt Instructions 26
11. RISC-V AES Decrypt Instructions Assembly Use Case 26
12. RISC-V SHA256 sig0 and sig1 Instructions Assembly Syntax 28
13. SAIL Pseudocode for RISC-V SHA256 sig0 and sig1 Instructions 28
14. RISC-V SHA256 sig0 and sig1 Instructions Assembly Use Case 28
15. RISC-V SHA256 sum0 and sum1 Instructions Assembly Syntax 29
16. SAIL Pseudocode for RISC-V SHA256 sum0 and sum1 Instructions 29
17. RISC-V SHA256 sum0 and sum1 Instructions Assembly Use Case 30
18. Pseudocode for Intel's AES Instructions 31
19. Intel's AES Instructions Assembly Use Case 32
20. RISC-V Vector AES Encrypt Instructions Assembly Syntax 37
21. SAIL Pseudocode for RISC-V Vector AES Encrypt Instructions 38
22. RISC-V Vector AES Encrypt Instructions Assembly Use Case 38
23. RISC-V Vector AES Decrypt Instructions Assembly Syntax 38
24. SAIL Pseudocode for RISC-V Vector AES Decrypt Instructions 39

25. RISC-V Vector AES Decrypt Instructions Assembly Use Case 40
26. RISC-V Vector SHA256 Scheduling Instruction Assembly Syntax 41
27. SAIL Pseudocode for RISC-V Vector SHA256 Scheduling Instruction 42
28. RISC-V Vector SHA256 Scheduling Instruction Assembly Use Case 42
29. RISC-V Vector SHA256 Hash Instruction Assembly Syntax 43
30. SAIL Pseudocode for RISC-V Vector SHA256 Hash Instruction 43
31. RISC-V Vector SHA256 Hash Instruction Assembly Use Case 44
32. Vivado Wave Window Depiction of Vector AES-128 Encryption 48
33. Vivado Wave Window Depiction of First AES Round 49
34. Vivado Wave Window Depiction of Last AES Round 49
35. Vivado Wave Window Depiction of Vector SHA256 Message Schedule Generation 52
36. Vivado Wave Window Depiction of First 8 SHA256 Double-Words 53
37. Vivado Wave Window Depiction of Last 8 SHA256 Double-Words 53

Chapter 1

INTRODUCTION

1.1 Statement of Problem

RISC-V is an open-source instruction set architecture (ISA) developed at UC Berkeley in 2010 [1]. The ISA was of years of years of reduced instruction set computer (RISC) development, with the goal of creating a global open-source engineering standard for microcontroller architecture. Development and improvement of the standard are completed through open involvement by members of the engineering community, which has resulted in the creation of over twenty extensions to the base instruction set [2].

In December of 2021, the RISC-V Foundation ratified the Vector and Scalar Cryptography Extensions to the RISC-V ISA [3]. This marked a huge step forward for the architecture, streamlining parallel data processing and execution of popular security algorithms. However, cryptography algorithms usually handle data widths larger than what the base RISC-V scalar architecture can handle, including 128 bits for AES and 256 bits for SHA256, two operations the Scalar Cryptography extension targets specifically [4][5][6]. Cryptography algorithms also feature significant data-level parallelism, especially when encrypting large amounts of data, that cannot be exploited with the scalar instructions.

One solution is to use the new Vector Extension to draft vector instructions that would better handle these large data widths and parallel processes. Commercial processors like those sold by Intel already use similar SIMD instructions for encryption solutions. Vector cryptography instructions could potentially be used to complete entire encryption rounds of a given algorithm in a single instruction, as opposed to the scalar cryptography instructions which target only one or two recurring equations. Assigning the increased data widths of cryptography algorithms to the vector portion of a RISC-V processor also allows the main microcontroller to keep the smaller standard 32- or 64-bit sizes that are acceptable for many other operations [7][8]. The RISC-V Foundation has noted their intension to add vector cryptography instructions to RISC-V themselves thorough future ratification of a Volume II to the RISC-V Cryptography Extension that

would include vector instructions, and had originally aimed to do so in Q1 2022, though it has not yet been completed [6].

This thesis poses the question: what new vector cryptography instructions should be added in a future extension? A new set of instructions must be detailed and tested. High-level cryptography functions must be broken down to observe how they may be implemented in hardware and defined by instructions in a way that is standardized, efficient, and an improvement on the current scalar instructions. This thesis will attempt to identify what vector cryptography instructions can be added to streamline the two most commonly used algorithms targeted by the original scalar cryptography extension: AES and SHA256. As with instructions in previous RISC-V Extensions, the new instructions will require a description of the instruction, a 32-bit format to be read by microcontrollers, pseudocode describing what operations the instruction will complete, and assembly-level code showing how the instruction may be used.

1.2 List of Terms

- AES: Advanced Encryption Standard. A block-cypher that encrypts data with a user-specified key. Can be decrypted using the same key.
- ALU: Arithmetic Logic Unit. A circuit in a computer's central processing unit that performs basic mathematical calculations.
- Byte: A unit of computer information or data-storage that consists of a group of eight bits.
- CPU: Computer Processing Unit. The component of a computer that performs the basic operations of the system, that exchanges data with the system's memory or peripherals, and that manages the system's other components.
- Cryptography: The enciphering and deciphering of messages in secret code or cipher. Specifically referring to modern-day security algorithms.
- Decoder: A circuit in the central processing unit in charge of interpreting what instructions are being read from memory.
- Double-word: Four bytes, or thirty-two bits of data or memory storage.
- Extension: An addition to a base ISA specifying new instructions or functionality.

- Hash: A value produced through mathematical functions that is unique to a certain set of data used for verification and authentication. The data which the hash refers to cannot be decoded from the hash itself.
- Instruction: Low-level, bit-mapped data stored in read-only memory that specifies a singular register-level function.
- ISA: Instruction Set Architecture. A standard set of register-level instructions that define hardware-level functionality of a microprocessor and how it can be controlled by software.
- RISC: Reduced Instruction Set Architecture. A computer processor designed to execute a small set of instructions quickly.
- Register: A read-write memory location in a CPU that holds a single set of data. Data length held by a register is determined by the CPU architecture but is usually 32 or 64 bits.
- Round: A set of mathematical equations that is repeated a set number of times during an encryption algorithm.
- SHA: Secure Hashing Algorithm. A standardized mathematical process for producing a hash for a given input.
- SIMD: Single-Instruction Multi-Data. Refers to instructions that perform functions on multiple data at once.
- Vector Register: A register that represents a one-dimensional array of data. Other instructions are used to specify the length of both the array and the data within, but the vector register itself does not contain this information. Functions are executed on each piece of data in a vector register simultaneously.
- VLEN: The length of vector registers in the RISC-V vector specification. Must be a power of 2.
- Word: Two bytes, or sixteen bits of data or memory storage.

1.3 Purpose of Study

This thesis aims to suggest meaningful vector cryptography instruction additions to the RISC-V architecture that are realizable in real hardware. AES and SHA will be the main focus of this thesis, as they are both popular algorithms backed by the National Institute of Technology for general use in today's tech industry, and are both already supported by the RISC-V Scalar Cryptography extension [6][9]. Any new instructions aim to follow the conventions established in existing RISC-V specifications, and to provide a performance and ease-of-use improvement over the scalar cryptography instruction set. 32-bit instruction formats will be suggested for all new instructions, and any hardware requirements to run the instructions will also be noted. Pseudocode will be provided for every suggested instruction to detail its functions, and an example using the instruction use will be written in RISC-V assembly code. This will ensure that any future contributors or users will understand how the instructions work and should be used.

This thesis does not aim to convince the reader of the benefits of SIMD instructions, the usefulness of cryptography instructions on RISC-V, or the role of cryptography in society at large, nor does it attempt to produce a complete Vector Cryptography extension. Future work will be necessary to tun the few instructions suggested here into a full standard ready for ratification.

To begin, The AES and SHA algorithms are examined step-by-step to identify parallel processes, repeated calculations, and other aspects that can be streamlined by the addition of vector calculations. RISC-V itself is examined to keep instruction suggestions within the scope of the goals, format, and limitations of the existing architecture. The RISC-V Vector Extension is looked at to note the limits placed on vector sizes and hardware, the types of instructions currently available, and the standardized instruction format. The RISC-V cryptography extension is referenced to understand which cryptography algorithms the RISC-V community find to be most important, and how these instructions might benefit from being expanded through vector implementation. Intel's own AES and SHA SIMD instructions are analyzed as a point of comparison. A RISC-V vector coprocessor, Vicuna, is used to test hardware limitations and observe how existing vector instructions handle data and execution [10].

This thesis suggests six vector cryptography instructions looking to streamline the AES and SHA 256 algorithms: vaesesm, vaeses, vaesdsm, vaesds, sha256sch, and sha256hash. Descriptions, instruction formats, pseudocode, and suggested assembly use of all six are provided. A Verilog implementation of vaesesm, vaeses, and vsha256sch follow, demonstrating a complete AES encryption and SHA256 message schedule generation in hardware. Instructions vaesds and vaesdsm are not implemented due to their computational similarity to vaeses and vaesesm, and vsha256hash is left out due to its architectural complexity when compared to the other instructions.

Chapter 2

LITERATURE REVIEW

2.1 Advanced Encryption Standard (AES)

A look at the math behind AES will help decide what kinds of vector instructions could benefit the process.

The Advanced Encryption Standard, also known as the Rijndael Cipher, was invented in 1998, and replaced the Data Encryption Standard (DES) in 2002, which at the time was severely outdated and easily cracked due to its short key length of 56 bits [11]. AES, in contrast, supports key lengths of 128, 192, or 256 bits, denoted by monikers of AES-128, AES-192, and AES-256, all of which have yet to be cracked [4]. Using 8-bit ASCII characters, the three key lengths allow passwords of sixteen, twenty-four, and thirty-two characters respectively, with longer keys being more secure.

AES is currently one of two block cyphers supported by the National Institute of Science and Technology, and is used in most data security channels including passwords, WIFI security, VPNs, compression tools, and operating system components [9][11].

AES encryption is byte-oriented and takes data input as 128-bit blocks. The data is encrypted in repeated processes, called rounds – ten for AES-128, twelve for AES-192, and fourteen for AES-256. Each round involves a unique 128-bit round key derived using a key-expansion algorithm from the original encryption key. A single encryption round is identical for all three variants, with the original key length only affecting the number of rounds of key-expansion and encryption/decryption, summarized in Table 1 [4].

Table 1: AES Type Information

AES Version	Encryption Key Length	Input Data Length	Number of Key Expansion Rounds	Number of Encryption/Decryption Rounds
AES-128	128	128	10	10
AES-192	192	128	9	12
AES-256	256	128	7	14

The 128-bit input is stored in memory as a column-major matrix of sixteen bytes, known as a state array [12]. The cipher output after all rounds is also 128 bits. The key-expansion algorithm, encryption rounds, and decryption rounds are described in the next three sections.

2.1.1 AES Key Schedule

The encryption key, decided by the user, is restricted to either 128, 192, or 256 bits as needed, and is input to a key-expansion algorithm that creates subsequent 128-bit round keys for each encryption round – ten for AES-128, twelve for AES-192, and fourteen for AES-256, plus a 0 round key taken directly from the first four words of the original key [4].

The key-expansion algorithm for a 128-bit encryption key follows the format in Figure 1, with the required double-words per round simply expanded to six for 192-bit and eight for 256-bit [12]. The full algorithm results in forty words for the 128-bit key, forty-eight for 192-bit, and fifty-six for 256-bit, that are then grouped by fours into the requisite round keys, summarized in Table 2.

Table 2: AES Key Expansion Information

AES Version	Encryption Key Length	Number of Key Expansion Rounds	Double-Words Generated per Expansion Round	Total Generated Double-Words	Round Keys Created
AES-128	128	10	4	40	10
AES-192	192	8	6	48	12
AES-256	256	7	8	56	14

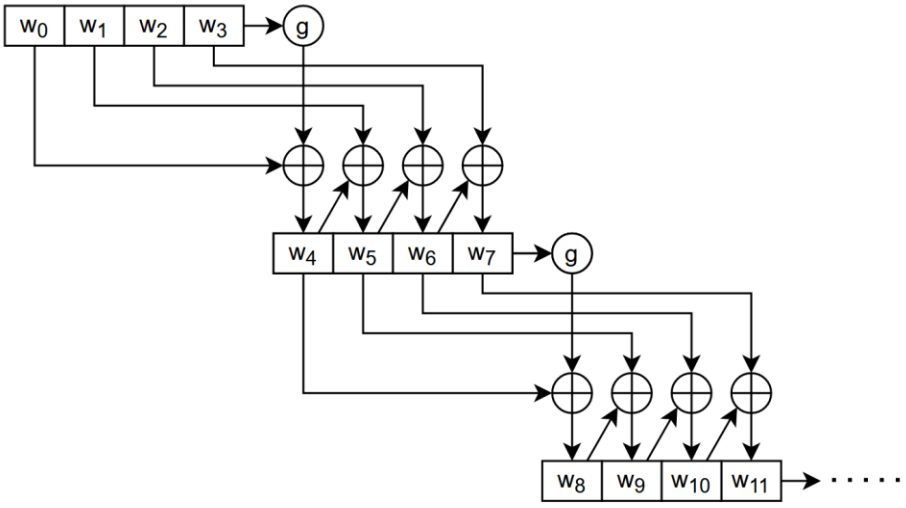

Figure 1: AES Key Schedule Generation Workflow

Each round of the key expansion generates the same number of words as are input. With AES-128 as an example, a round consists of taking the previous four words w_i, w_{i+1}, w_{i+2} and w_{i+3} as input. The first generated word, w_{i+4}, is calculated using equation (1) [12].

$$w_{i+4} = w_i \oplus g(w_{i+3}) \quad (1)$$

where:

$$g(X) = Sbox\big(ROL(X, 8)\big) \oplus \{RC_i, 0x00, 0x00, 0x00\}$$

and:

$$RC_1 = 0x01$$

$$RC_i = 0x02 * RC_{i-1}$$

The Sbox operation in g() is a Rijndael byte-swap that will be discussed in detail in Section 2.1.2. The last three words generated per round are derived with equations (2), (3), and (4) by XOR'ing previous words [12].

$$w_{i+5} = w_{i+4} \oplus w_{i+1} \quad (2)$$

$$w_{i+6} = w_{i+5} \oplus w_{i+2} \quad (3)$$

$$w_{i+7} = w_{i+6} \oplus w_{i+4} \quad (4)$$

This process of generating four words from the previous four continues until forty words have been created for AES-128, in addition to the original four words for a total of forty-four.

While vector instructions could speed up the generation of the round keys due to the vector registers' ability to hold the four double-words needed every round, it should not be a priority instruction. Generation of the key schedule only happens once, and can be used in as many encryptions as the user sees fit until they change the original encryption key. Likewise, speeding up key scheduling is not a current feature of the 32-bit scalar cryptography instructions this thesis is looking to adapt, so vector implementation of this portion of AES will be left to future work.

2.1.2 Encryption

As encryption for AES-128, -192, and -256 are identical save for the number of rounds, AES-128 will be referenced exclusively in this section.

Encryption begins with bitwise-adding round key 0 (the first four key words) to the input 128-bit block, followed by ten encryption rounds, shown in Figure 2 [12]. Each round consists of the same four steps: Substitute Bytes, Shift Rows, Mix Columns, and Add Round Key [4].

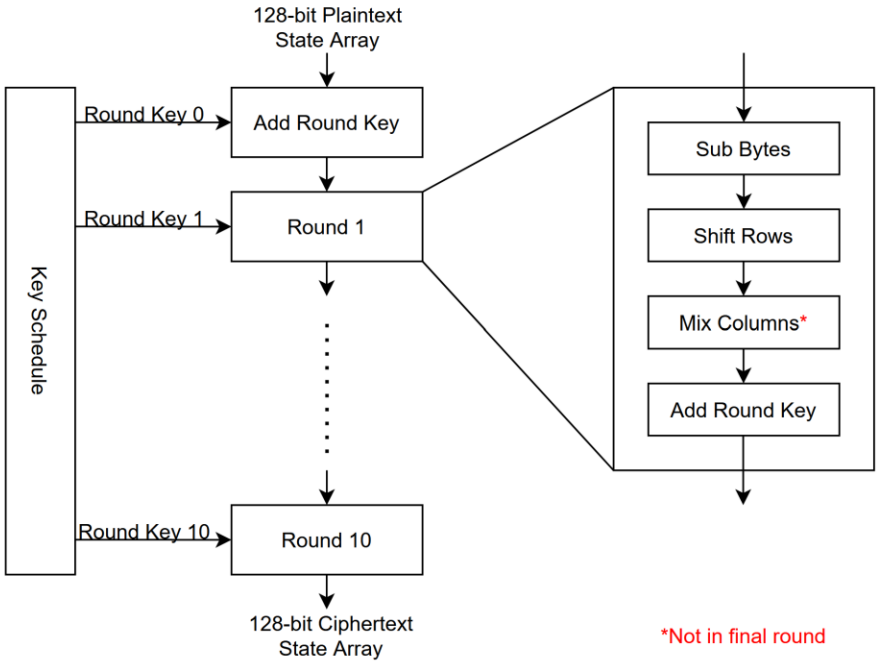

Figure 2: AES Encryption Workflow

The Substitute Bytes step consists of substituting every byte in the input block with that of the corresponding byte in the Rijndael Substitution Box (S-box). The S-box, splitting the input and output bytes into eight input bits “b” and eight output bits “s,” is calculated using the affine transformation in equation (5) [13].

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \\ s_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad (5)$$

However, this calculation is usually simplified to a lookup table. In the S-Box of Table 3, the least significant nibble of each input byte determines the column of substitution, and the most significant nibble determines the row [13]. Once all sixteen bytes have been substituted in this fashion, the step is complete.

Table 3: AES Substitution Lookup Table

	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

The next step, Shift Rows, circularly shifts the second row of the substituted state array one byte to the left, the third row two bytes to the left, and the fourth row three bytes to the left, shown in equation (6) [12].

$$\begin{bmatrix} s_{0.0} & s_{0.1} & s_{0.2} & s_{0.3} \\ s_{1.0} & s_{1.1} & s_{1.2} & s_{1.3} \\ s_{2.0} & s_{2.1} & s_{2.2} & s_{2.3} \\ s_{3.0} & s_{3.1} & s_{3.2} & s_{3.3} \end{bmatrix} \Longrightarrow \begin{bmatrix} s_{0.0} & s_{0.1} & s_{0.2} & s_{0.3} \\ s_{1.1} & s_{1.2} & s_{1.3} & s_{1.0} \\ s_{2.2} & s_{2.3} & s_{2.0} & s_{2.1} \\ s_{3.3} & s_{3.0} & s_{3.1} & s_{3.2} \end{bmatrix} \tag{6}$$

Mix Columns step computes the determinant in equation (7) with the output of the Shift Rows [12]. This step is not performed in the final round.

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \times \begin{bmatrix} s_{0.0} & s_{0.1} & s_{0.2} & s_{0.3} \\ s_{1.1} & s_{1.2} & s_{1.3} & s_{1.0} \\ s_{2.2} & s_{2.3} & s_{2.0} & s_{2.1} \\ s_{3.3} & s_{3.0} & s_{3.1} & s_{3.2} \end{bmatrix} = \begin{bmatrix} s'_{0.0} & s'_{0.1} & s'_{0.2} & s'_{0.3} \\ s'_{1.0} & s'_{1.1} & s'_{1.2} & s'_{1.3} \\ s'_{2.0} & s'_{2.1} & s'_{2.2} & s'_{2.3} \\ s'_{3.0} & s'_{3.1} & s'_{3.2} & s'_{3.3} \end{bmatrix} \tag{7}$$

Finally, Add Round Key XORs the output of the Mix Columns step with the next round key. This set of steps (excluding Mix Columns in the final round) is repeated until all rounds have completed and the input is fully encrypted.

The 128-bit inputs and round keys lend themselves well to the increased data width offered by vector registers. Substitute Bytes, Shift Rows, Mix Columns, and round key addition steps could be completed together for the full input array, as all necessary data for a given round would be available. This makes AES encryption rounds a good choice for vector integration. For vector register widths larger than 128-bits, multiple encryptions could even be done in parallel – though not multiple rounds of the same encryption, as they are completed serially.

2.1.3 Decryption

As with encryption, decryption for AES-128, -192, and -256 are identical save for the number of rounds. AES-128 again will be referenced exclusively in this section for simplicity.

Decryption uses the same key schedule as encryption, but in reverse. Decryption begins with adding round key 10 (the last four key words) to the input 128-bit block, followed by ten decryption rounds, shown in Figure 3. Each round consists of the same four steps as encryption, but inverted and in a different order: Inverse Shift Rows, Inverse Substitute Bytes, Add Round Key, and Inverse Mix Columns [4].

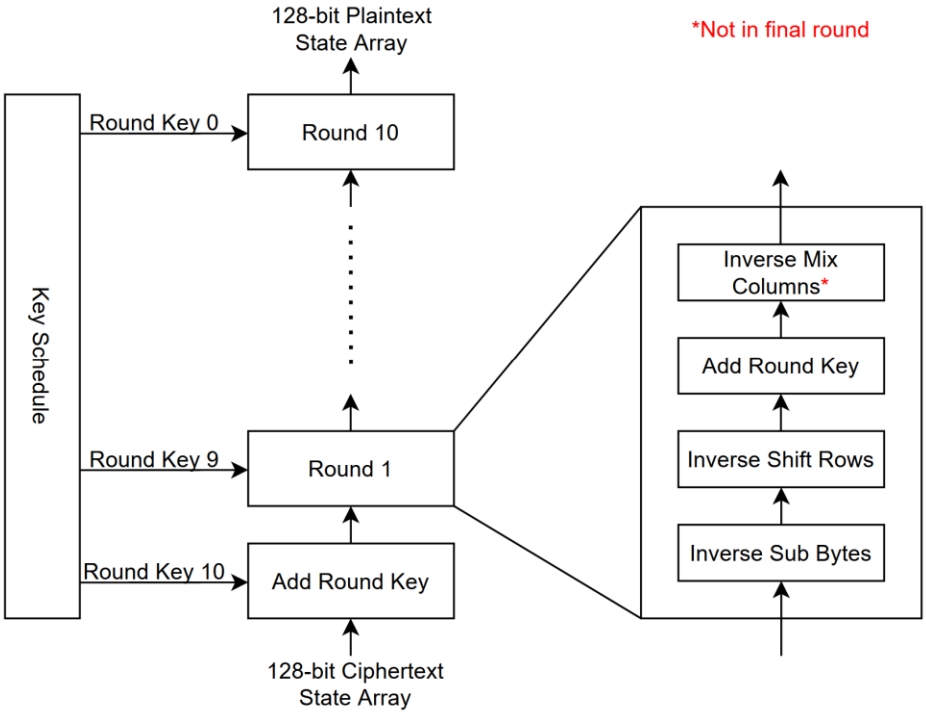

Figure 3: AES Decryption Workflow

The decryption steps are nearly identical to encryption, but inversed. Inverse Shift Rows rotates the second row one byte to the right, the third row two bytes to the right, and the fourth row three bytes to the right, shown in equation (8) [12].

$$\begin{bmatrix} s_{0.0} & s_{0.1} & s_{0.2} & s_{0.3} \\ s_{1.0} & s_{1.1} & s_{1.2} & s_{1.3} \\ s_{2.0} & s_{2.1} & s_{2.2} & s_{2.3} \\ s_{3.0} & s_{3.1} & s_{3.2} & s_{3.3} \end{bmatrix} \Rightarrow \begin{bmatrix} s_{0.0} & s_{0.1} & s_{0.2} & s_{0.3} \\ s_{1.3} & s_{1.0} & s_{1.1} & s_{1.2} \\ s_{2.2} & s_{2.3} & s_{2.0} & s_{2.1} \\ s_{3.1} & s_{3.2} & s_{3.3} & s_{3.0} \end{bmatrix} \tag{8}$$

Inverse Substitute Bytes uses the Inverse Rijndael S-Box in Table 4 [13].

Table 4: AES Inverse Substitution Lookup Table

	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

Inverse Mix Columns uses a different determinant, shown in equation 9. [12].

$$\begin{bmatrix} e & b & d & 9 \\ 9 & e & b & d \\ d & 9 & e & b \\ b & d & 9 & e \end{bmatrix} \times \begin{bmatrix} s_{0.0} & s_{0.1} & s_{0.2} & s_{0.3} \\ s_{1.3} & s_{1.0} & s_{1.1} & s_{1.2} \\ s_{2.2} & s_{2.3} & s_{2.0} & s_{2.1} \\ s_{3.1} & s_{3.2} & s_{3.3} & s_{3.0} \end{bmatrix} = \begin{bmatrix} s'_{0.0} & s'_{0.1} & s'_{0.2} & s'_{0.3} \\ s'_{1.0} & s'_{1.1} & s'_{1.2} & s'_{1.3} \\ s'_{2.0} & s'_{2.1} & s'_{2.2} & s'_{2.3} \\ s'_{3.0} & s'_{3.1} & s'_{3.2} & s'_{3.3} \end{bmatrix} \quad (9)$$

And finally, Add Round Key reverses the key schedule order. The output is a 128-bit decrypted data block.

AES decryption would benefit from vector instructions for the same reason as encryption: full state arrays could fit in a single register, allowing rounds to be completed in full, and in parallel with other decryptions if register width allowed.

2.2 Secure Hashing Algorithm 2 (SHA2)

A look at the SHA256 algorithm will help decide what functions to target with vector instructions.

Secure Hashing Algorithm 2 (SHA2) is a set of cryptographic hash functions, primarily referring to SHA256 and SHA512 [14]. Unlike encryption, hashing algorithms produce outputs that cannot be decrypted. SHA2 is currently used in TLS and SSL security protocols, and is the method of choice for verifying transactions in popular blockchain applications, including Bitcoin [14]. SHA256 is used for generating 256-bit hashes, and SHA512 is used for generating 512-bit hashes. The length of the output hash does not depend on input message length [5].

SHA256 and SHA512 are similar, with the former using 512-bit block inputs and 256-bit outputs, and the latter using 1024-bit block inputs and 512-bit outputs. SHA256 is also primarily computed through 32-bit constants and features fewer rounds than SHA512, which uses 64-bit constants in its computation of the output [5]. For simplicity, SHA256 will be the focus of this section.

2.2.1 Secure Hashing Algorithm 256 (SHA256)

The SHA256 algorithm calculates a 256-bit hash from any message up to 2^{64} bits in total length. The message must be broken into input blocks of 512-bits each, which are hashed sequentially. The input of 512 bits being reduced to an output of 256 bits is one of the features that makes the algorithm irreversible. The final input block must also be 512 bits, even if the original input message is not a factor of 512, and is thus padded to ensure correct length. Padding of the final block involves appending a 1, followed by 0's until the block length is 448 bits. The bit-length of the total message is then appended as a 64-bit value, to create a final block length of 512 bits. If the final block is less than 512 bits but more than 448, the final message is appended with a 1 and followed by 0's until it is 512 bits, and an extra block is added to the message that is 448 0's plus the 64-bit total message length [5]. The block separation and padding process is summarized in Figure 4, where L is the message length [15].

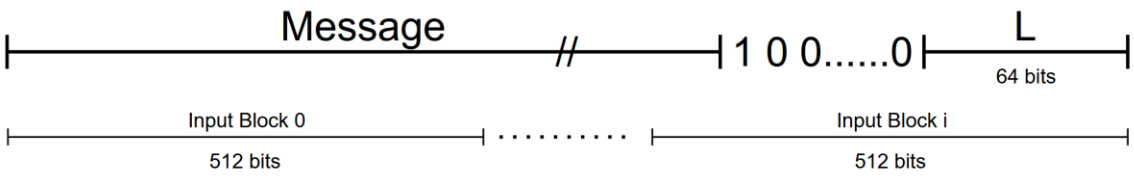

Figure 4: SHA256 Message Decomposition and Padding

Two tables of values are initialized. Sixty-four 32-bit standardized round constants K_i (given by the 32 first bits of the fractional parts of the cube roots of the first sixty-four prime numbers) are defined for each hashing round, and 8 32-bit standard initial hash values H_1^0 through H_8^0 (given by the first 32 bits of the fractional part of the square roots of the first eight prime numbers) are necessary to give the hashing process starting values. H_1^0 through H_8^0 are only necessary for the first block hashed, as initial hash values of every subsequent round will be the output hash values of the previous round [5]. The round keys and initial hashing values are listed in Tables 5 and 6.

Table 5: SHA256 Round Constants

428a2f98	71374491	b5c0fbcf	e9b5dba5	3956c25b	59f111f1	923f82a4	ab1c5ed5
d807aa98	12835b01	243185be	550c7dc3	72be5d74	80deb1fe	9bdc06a7	c19bf174
e49b69c1	efbe4786	0fc19dc6	240ca1cc	2de92c6f	4a7484aa	5cb0a9dc	76f988da
983e5152	a831c66d	b00327c8	bf597fc7	c6e00bf3	d5a79147	06ca6351	14292967
27b70a85	2e1b2138	4d2c6dfc	53380d13	650a7354	766a0abb	81c2c92e	92722c85
a2bfe8a1	a81a664b	c24b8b70	c76c51a3	d192e819	d6990624	f40e3585	106aa070
19a4c116	1e376c08	2748774c	34b0bcb5	391c0cb3	4ed8aa4a	5b9cca4f	682e6ff3
748f82ee	78a5636f	84c87814	8cc70208	90befffa	a4506ceb	bef9a3f7	c67178f2

Table 6: SHA256 Initial Hash Values

Hash Double-Word	Initial Value
H_1^0	6a09e667
H_2^0	bb67ae85
H_3^0	3c6ef372
H_4^0	a54ff53a
H_5^0	510e527f
H_6^0	9b05688c
H_7^0	1f83d9ab
H_8^0	5be0cd19

The padded input message is hashed one 512-bit block at a time. The hashing process is summarized in Figure 5.

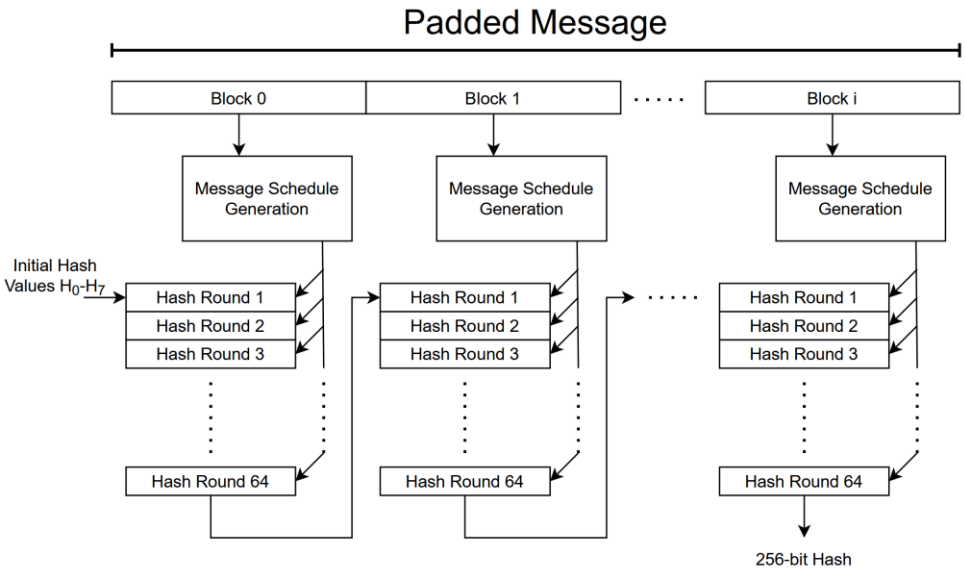

Figure 5: SHA256 Hashing Workflow

When a block begins hashing, a message schedule of sixty-four double-words W_i of 32 bits each is generated. The first sixteen double-words W_1 through W_{16} of the schedule are formed

by splitting the 512-bit input block into 32-bit chunks [5]. The next forty-eight are calculated in equation (10), with σ_0 and σ_1 referred to as the Sigma 0 and Sigma 1 functions.

$$W_i = \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16}, \qquad 17 \leq i \leq 64 \tag{10}$$

where:

$$\sigma_0(X) = ROR(X, 7) \oplus ROR(X, 18) \oplus SLR(X, 3)$$
$$\sigma_0(X) = ROR(X, 17) \oplus ROR(X, 19) \oplus SLR(X, 10)$$

64 hashing rounds are then performed. Each round, intermediate values a, b, c, d, e, f, g and h in equation (11) are defined as the previous round's hash values, starting with H_1^0 through H_8^0 in the first hash's first round [5].

$$a_{i-1},\ b_{i-1},\ c_{i-1},\ d_{i-1},\ e_{i-1},\ f_{i-1},\ g_{i-1},\ h_{i-1} = H_1^{i-1},,\ H_2^{i-1},\ H_3^{i-1},\ H_4^{i-1},\ H_5^{i-1},\ H_6^{i-1},\ H_7^{i-1},\ H_8^{i-1} \tag{11}$$

The intermediate values a, b, c, d, e, f, g and h for the previous round, the round constant K_i, and the message schedule word W_i are then used to calculate intermediate values for the next round "i", described by equation (12), with Σ_0 and Σ_1 referred to as the Sum 0 and Sum 1 functions [5].

$$T_1 = h + \Sigma_1(e_{i-1}) + CH(e_{i-1},\ f_{i-1},\ g_{i-1}) + K_i + W_i \tag{12}$$
$$T_2 = \Sigma_0(a_{i-1}) + MAJ(a_{i-1},\ b_{i-1},\ c_{i-1})$$
$$h = g_{i-1}$$
$$g = f_{i-1}$$
$$f = e_{i-1}$$
$$e = d_{i-1} + T_1$$
$$d = c_{i-1}$$
$$c = b_{i-1}$$
$$b = a_{i-1}$$
$$a = T_1 + T_2$$

where:

$$CH(X, Y, Z) = (X \,\&\, Y) \oplus (\sim X \,\&\, Z)$$
$$MAJ(X, Y, Z) = (X \,\&\, Y) \oplus (X \,\&\, Z) \oplus (Y \,\&\, Z)$$
$$\Sigma_0(X) = ROR(X, 2) \oplus ROR(X, 13) \oplus ROR(X, 22)$$
$$\Sigma_1(X) = ROR(X, 6) \oplus ROR(X, 11) \oplus ROR(X, 25)$$

Finally, the new intermediate values are added to the previous round's hash values to create new hash values, as shown in equation (13). This encompasses a single SHA256 round [5].

$$
\begin{aligned}
H_1^i &= H_1^{i-1} + a \\
H_1^i &= H_2^{i-1} + b \\
H_1^i &= H_3^{i-1} + c \\
H_1^i &= H_4^{i-1} + d \\
H_1^i &= H_5^{i-1} + e \\
H_1^i &= H_6^{i-1} + f \\
H_1^i &= H_7^{i-1} + g \\
H_1^i &= H_8^{i-1} + h
\end{aligned}
\tag{13}
$$

After all sixty-four rounds have completed, the final eight hash values are concatenated to create the 256-bit hash output in equation (14). If more blocks of the input message remain, these eight hash values are used as initial hash values H_1^0 through H_8^0 in the next block's hash [5].

$$H = \{H_1^i,\ H_2^i,\ H_3^i,\ H_4^i,\ H_5^i,\ H_6^i,\ H_7^i,\ H_8^i\} \tag{14}$$

The two overarching areas of SHA256 hashing that could benefit from vector instructions are the generation of the message schedule and the hashing rounds themselves. Unlike AES, the SHA message schedule changes for every new input. While not performed as frequently as hashing rounds, the message schedule generation would still be repeated for as many times as there are inputs, make it a clear candidate for a vector instruction.

Hashing rounds, being ran sixty-four times per hash, are an obvious choice for a dedicated vector instruction. Because hashing rounds are ran serially and produce the input for the next hash, they could not be completed in parallel with other hashing rounds, unless entirely unrelated data is being hashed. Theses rounds could still make use of vector registers' increased width to handle the full 256-bit intermediate hashing values at once.

2.3 RISC-V

To implement new instructions on the RISC-V architecture, its current hardware abilities and standards must be examined so that the proposed instructions will meet the needs and expectations of existing RISC-V microcontrollers and the development community behind them. Reduced Instruction Set Computer V, or RISC-V, is a leading open-source instruction-set architecture (ISA) developed in 2010, currently overseen by the RISC-V Foundation [1]. The RISC-V ISA is open-source and has been widely embraced in recent years for research and academia, streamlining the development of experimental processors. Goals of the RISC-V ISA outlined by the Foundation, and subsequently goals supported by this thesis, include a free and open ISA description, functionality in real hardware implementation, avoidance of over-architecting, and avoidance of implementation details where possible [2].

RISC-V features a base integer ISA in 32- and 64-bit register and address space variants, termed RV32I and RV64I, both featuring thirty-two general-purpose registers x0-x31. The base ISAs are kept simple to allow a high level of customizability [2].
Functionality outside of basic integer operations are performed through the addition of ISA extensions, both in the codebase and hardware. ISA Extensions are developed jointly by the community and go through an extensive review process before being officially frozen and ratified by the Foundation. Current ratified instruction extensions include Integer Multiplication (M), Atomic (A), Control and Status Register (Zicsr), Single-, Double-, and Quad-Precision Floating-Point (F, D, and Q), Decimal Floating-Point (L), Compressed (C), Bit Manipulation (B), Dynamically Translated Languages (J), Transactional Memory (T), Packed-SIMD (P), Vector (V), and Cryptography (K) [2]. RISC-V processors that include any of the previously listed extensions are identified by appending the extension letter to the end of the base integer ISA – for example, adding integer multiplication and floating-point functionality to the base 32-bit ISA would result in an identifier of RV32IMF.

The new instructions proposed in this thesis aim to be a combination of the format of the Vector extension with the functionality of the Cryptography extension. Both will be examined in

the next two sections to observe what instructions are currently available, how they're implemented, and how new instructions may be integrated into these standards.

2.3.1 RISC-V Vector Extension

The Vector extension for RISC-V was ratified on December 2, 2021, adding thirty-two vector registers to the specification [3]. Vector operations allow single instructions to perform operations on multiple sets of data, much like arrays in higher level programming. The specification defines a vector register width, VLEN, as a power of 2 with a maximum width of 2^{16} bits [16].

Vector instruction formats are 32 bits wide, and are grouped under four categories: vector load, vector store, vector arithmetic, and vector configuration. Vector arithmetic instructions are further classified as integer, floating-point, or masked, and compose seven instruction types when paired with available operand combinations [16]. Because any potential vector cryptography instructions examined in this thesis would fall under the vector-arithmetic format with integer operands, this is the only type that will be examined in detail.

The three types of integer arithmetic instructions, vector-vector (OPIVV), vector-immediate (OPIVI), and vector-scalar (OPIVX), share formats summarized in Table 7.

Table 7: Formats for Vector Integer Arithmetic Instructions

Instruction Type	FUNC6						VM	VS2					Operand 1					FUNC3			VD							OPCODE						
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OPIVV														VS1				0	0	0						1	0	1	0	1	1	1
OPIVI														IMM				0	1	1						1	0	1	0	1	1	1
OPIVX														RS1				1	0	0						1	0	1	0	1	1	1

Bits [6:0] are always the vector opcode of {1010111}. Bits {11:7} are usually the destination register for storing results, but occasionally represent a third operand for ternary instructions. Bits {14:12} are func3 and define what type of vector integer arithmetic instruction is being called. Bits {19:15} define operand one, and can be a vector register, scalar register, or immediate depending on the instruction type. Bits {24:20} are the second operand and are always

a vector register. Bit {25} is the masking bit, set high if normal and low if masked. Bits {31:26} are the func6 code that define specifically what vector instruction is being called from the func3 type category [16].

These instruction formats set certain limitations on the possible format and what is achievable with new instructions. At maximum, three inputs can be taken per instruction. The opcode {1010111} should be used to fit in with all other vector instructions and make integration to the existing extension easy. New func6 values must be unique to not overlap with decoding of any existing instructions. Operands will be vector-vector, so the func3 value will be locked to {000}. Evaluation of the Scalar Cryptography extension will further define the format of RISCV vector cryptography instructions.

2.3.2 RISC-V Scalar Cryptography Extension

The RISC-V Scalar Cryptography Extensions Volume I: Scalar & Entropy Source Instructions specification was ratified on December 2, 2021 [3]. The specification describes scalar cryptography instructions for both the 32- and 64-bit base architectures and provides architectural interface to an Entropy Source. Volume II, not yet published, will describe Vector Cryptography instructions for the 32- and 64-bit architectures, but was previously waiting on ratification of the base Vector extension [6].

The specification is aimed primarily at cryptographers and cryptographic software developers who would benefit most of the addition of cryptography instructions, but also the computer architects, digital design engineers, and verification engineers who would be responsible for their implementation [6].

Design of the RISC-V Scalar Cryptography extension follows four core policies. First, the cryptography extension will always prefer to target algorithm performance generalizability to other use cases. Second, the extension only aims to support current cryptographic algorithms, and does not aim to implement instructions for proposed or theoretical algorithms. Third, the extension will not attempt to implement low-level instructions that may be useful for future, unforeseen standards. Finally, the extension must aim to remove the possibility of timing side-

channels, and therefore will not provide support for timing countermeasures implemented above the ISA [6].

Four primary cryptographic functions are targeted by the scalar cryptography extension: Advanced Encryption Standard (AES), Secure Hashing Algorithm 2 (SHA2), ShangMi 3 (SM3), and Shang Mi 4 (SM4). SHA2 and AES respectively represent the hashing and encryption algorithms supported by the National Institute of Science and Technology (NIST), while SM3 and SM4 are respectively hashing and encryption algorithms used primarily for certain Chinese standards. The SHA2 and AES algorithms and their associated instructions will be the focus of this thesis due to the standards' common usage worldwide.

2.3.2.1 RISC-V Scalar AES Instructions

The RISC-V Scalar Cryptography Extension provides eleven instructions for AES; four for encryption, four for decryption, and three for the key schedule. The encryption suite has 32-bit instructions for the middle and final rounds of encryption (aes32esmi and aes32esi) and two 64-bit equivalent middle and final round instructions (aes64esm and aes64es). Decryption features the inverses of the same instructions (aes32dsmi, aes32dsi, aes64dsm, aes64ds). Two instructions for key schedule generation (aes64ks1i and es64ks2) are used for both encryption and decryption, and a decrypt key schedule mix columns instruction (aes64im) is provided exclusively for decryption [6]. The 32-bit instructions aes32esmi, aes32esi, aes32dsmi, and aes32dsi will be exclusively analyzed for vector implementation in this thesis.

The 32-bit RISC-V scalar AES instructions follow the formats in Table 8 and use 32-bit operands. Each instruction executes on a single byte of input register RS2, selected using the BS bits. Register RS1 holds one column of the round key, and register RD stores the 32-bit result.

Table 8: Formats for 32-bit Scalar Cryptography AES Instructions

Instruction	BS		FUNC5					RS2					RS1					FUNC3			RD					OPCODE						
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
aes32dsi			1	0	1	0	1											0	0	0						0	1	1	0	0	1	1
aes32dsmi			1	0	1	1	1											0	0	0						0	0	1	0	0	1	1
aes32esi			1	0	0	0	1											0	0	0						0	0	1	0	0	1	1
aes32esmi			1	0	0	1	1											0	0	0						0	0	1	0	0	1	1

2.3.2.1.1 AES32ESMI and AES32ESI

Instructions aes32esmi and aes32esi perform the middle and final rounds respectively of AES encryption operations on a single byte of input RS2. The assembly implementations of these two instructions follow the syntax in Figure 6.

```
aes32esmi rd rs1 rs2 bs
aes32esi rd rs1 rs2 bs
```

Figure 6: RISC-V AES Encrypt Instructions Assembly Syntax

Bits BS are used to select the input byte, which undergoes a Rijndael S-Box transformation, as described in Section 2.1.2. If running aes32esmi, a partial Mix Columns operation is then performed on the byte. Because there is no indication of byte placement in the state matrix, the byte is multiplied by 0x01, 0x01, 0x02, and 0x03 to account for all 4 possible multipliers during the usual Mix Columns step. This partial Mix Columns also ensures a 32-bit output. Instruction aes32esi does not perform this step. The set of bytes is then rotated left by BS*8 bits, XOR'ed with the round key column in RS1 and stored in RD. The pseudocode in Figure 7 summarizes the process of both instructions [6].

```
function clause execute (AES32ESMI (bs,rs2,rs1,rd)) = {
        let shamt : bits( 5) = bs @ 0b000; /* shamt = bs*8 */
        let si : bits( 8) = (X(rs2)[31..0] >> shamt)[7..0]; /* SBox Input */
        let so : bits( 8) = aes_sbox_fwd(si);
        let mixed : bits(32) = aes_mixcolumn_byte_fwd(so);
        let result : bits(32) = X(rs1)[31..0] ^ rol32(mixed, unsigned(shamt));
        X(rd) = EXTS(result); RETIRE_SUCCESS
}

function clause execute (AES32ESI (bs,rs2,rs1,rd)) = {
        let shamt : bits( 5) = bs @ 0b000; /* shamt = bs*8 */
        let si : bits( 8) = (X(rs2)[31..0] >> shamt)[7..0]; /* SBox Input */
        let so : bits(32) = 0x000000 @ aes_sbox_fwd(si);
        let result : bits(32) = X(rs1)[31..0] ^ rol32(so, unsigned(shamt));
        X(rd) = EXTS(result); RETIRE_SUCCESS
}
```

Figure 7: SAIL Pseudocode for RISC-V AES Encrypt Instructions

A full round of AES encryption requires sixteen calls of aes32esmi or aes32esi – one for every byte of the 128-bit input block. The assembly code in Figure 8 describes the standard assembly use case for middle and final rounds, where registers a0 through a3 initially hold the four round key columns and registers t0 through t3 initially hold the four input state matrix columns [6].

Middle Round	Last Round
aes32esmi a0, a0, t0, 0	aes32esi a0, a0, t0, 0
aes32esmi a0, a0, t1, 1	aes32esi a0, a0, t1, 1
aes32esmi a0, a0, t2, 2	aes32esi a0, a0, t2, 2
aes32esmi a0, a0, t3, 3	aes32esi a0, a0, t3, 3
aes32esmi a1, a1, t1, 0	aes32esi a1, a1, t1, 0
aes32esmi a1, a1, t2, 1	aes32esi a1, a1, t2, 1
aes32esmi a1, a1, t3, 2	aes32esi a1, a1, t3, 2
aes32esmi a1, a1, t0, 3	aes32esi a1, a1, t0, 3
aes32esmi a2, a2, t2, 0	aes32esi a2, a2, t2, 0
aes32esmi a2, a2, t3, 1	aes32esi a2, a2, t3, 1
aes32esmi a2, a2, t0, 2	aes32esi a2, a2, t0, 2
aes32esmi a2, a2, t1, 3	aes32esi a2, a2, t1, 3
aes32esmi a3, a3, t3, 0	aes32esi a3, a3, t3, 0
aes32esmi a3, a3, t0, 1	aes32esi a3, a3, t0, 1
aes32esmi a3, a3, t1, 2	aes32esi a3, a3, t1, 2
aes32esmi a3, a3, t2, 3	aes32esi a3, a3, t2, 3

Figure 8: RISC-V AES Encrypt Instructions Assembly Use Case

To adapt these instructions to the vector format, the most obvious advancement is to condense AES encryption rounds into a single instruction to avoid the repeated calls and reduce

space used in instruction memory. Like the scalar extension, a separate instruction for middle and final rounds is needed to omit the mix columns step.

2.3.2.1.2 AES32DSMI and AES32DSI

Instructions aes32dsmi and aes32dsi perform the middle and final rounds respectively of AES decryption operations on a single byte of input RS2. The assembly implementations of these two instructions follow the syntax in Figure 9.

```
aes32dsmi rd rs1 rs2 bs
aes32dsi rd rs1 rs2 bs
```

Figure 9: RISC-V AES Decrypt Instructions Assembly Syntax

Bits BS are used to select the input byte, which undergoes an Inverse Rijndael S-Box transformation, as described in Section 2.1.3. If running aes32dsmi, a partial Inverse Mix Columns operation is then performed on the byte. Because there is no indication of byte placement in the state matrix, the byte is multiplied by 0x0B, 0x0D, 0x09, and 0x0E to account for all 4 possible multipliers during the usual Inverse Mix Columns step. This partial Inverse Mix Columns also ensures a 32-bit output. Instruction aes32dsi does not perform this step. The set of bytes is then rotated left by BS*8 bits, XOR'ed with the round key column in RS1 and stored in RD. The pseudocode in Figure 10 summarizes the process of both instructions [6].

```
function clause execute (AES32DSMI (bs,rs2,rs1,rd)) = {
       let shamt : bits( 5) = bs @ 0b000; /* shamt = bs*8 */
       let si : bits( 8) = (X(rs2)[31..0] >> shamt)[7..0]; /* SBox Input */
       let so : bits( 8) = aes_sbox_inv(si);
       let mixed : bits(32) = aes_mixcolumn_byte_inv(so);
       let result : bits(32) = X(rs1)[31..0] ^ rol32(mixed, unsigned(shamt));
       X(rd) = EXTS(result); RETIRE_SUCCESS
}

function clause execute (AES32DSI (bs,rs2,rs1,rd)) = {
       let shamt : bits( 5) = bs @ 0b000; /* shamt = bs*8 */
       let si : bits( 8) = (X(rs2)[31..0] >> shamt)[7..0]; /* SBox Input */
       let so : bits(32) = 0x000000 @ aes_sbox_inv(si);
       let result : bits(32) = X(rs1)[31..0] ^ rol32(so, unsigned(shamt));
       X(rd) = EXTS(result); RETIRE_SUCCESS
}
```

Figure 10: SAIL Pseudocode for RISC-V AES Decrypt Instructions

A full round of AES decryption requires sixteen calls of aes32dsmi or aes32dsi – one for every byte of the 128-bit input block. The assembly code in Figure 11 describes the standard assembly use case for middle and final rounds, where registers a0 through a3 initially hold the four round key columns and registers t0 through t3 initially hold the four input state matrix columns [6].

Middle Round	Last Round
aes32dsmi a0, a0, t0, 0	aes32dsi a0, a0, t0, 0
aes32dsmi a0, a0, t1, 1	aes32dsi a0, a0, t1, 1
aes32dsmi a0, a0, t2, 2	aes32dsi a0, a0, t2, 2
aes32dsmi a0, a0, t3, 3	aes32dsi a0, a0, t3, 3
aes32dsmi a1, a1, t1, 0	aes32dsi a1, a1, t1, 0
aes32dsmi a1, a1, t2, 1	aes32dsi a1, a1, t2, 1
aes32dsmi a1, a1, t3, 2	aes32dsi a1, a1, t3, 2
aes32dsmi a1, a1, t0, 3	aes32dsi a1, a1, t0, 3
aes32dsmi a2, a2, t2, 0	aes32dsi a2, a2, t2, 0
aes32dsmi a2, a2, t3, 1	aes32dsi a2, a2, t3, 1
aes32dsmi a2, a2, t0, 2	aes32dsi a2, a2, t0, 2
aes32dsmi a2, a2, t1, 3	aes32dsi a2, a2, t1, 3
aes32dsmi a3, a3, t3, 0	aes32dsi a3, a3, t3, 0
aes32dsmi a3, a3, t0, 1	aes32dsi a3, a3, t0, 1
aes32dsmi a3, a3, t1, 2	aes32dsi a3, a3, t1, 2
aes32dsmi a3, a3, t2, 3	aes32dsi a3, a3, t2, 3

Figure 11: RISC-V AES Decrypt Instructions Assembly Use Case

As with the encryption instructions, the most obvious advancement to be made with vector instructions is to condense the decryption rounds into a single instruction to avoid the

repeated calls and reduce space used in instruction memory. A separate instruction for middle and final rounds is needed to omit the mix columns step.

2.3.2.2 RISC-V Scalar SHA2 Instructions

The RISC-V Scalar Cryptography Extension provides fourteen instructions for SHA2; four for SHA256 and seven for SHA512. The four SHA256 instructions for the 256-bit Sigma 0, Sigma 1, Sum 0 and Sum 1 functions (sha256sig0, sha256sig1, sha256sum0, and sha256sum1) are viable for either 32-bit or 64-bit base architectures. Due to larger operand size, sha512sig0h, sha512sig0l, sha512sig1h, and sig512sig1l are exclusive to the 32-bit architecture and perform low or high half of the 512-bit Sigma 0 and Sigma 1 functions. Instructions sha512sum0r and sha512sum1r perform half of the 512-bit Sum 0 and Sum 1 functions and must be ran twice each for the full process. The 64-bit exclusive instructions perform full 512-bit Sigma 0, Sigma 1, Sum 0 and Sum 1 functions (sha512sig0, sha512sig1, sha512sum0, and sha512sum1) [6]. The 32-bit SHA256 instructions will be analyzed for this thesis.

The 32-bit RISC-V scalar SHA256 instructions follow the formats in Table 9 and utilize 32-bit operands. Each instruction executes on the data in register RS1, and register RD stores the 32-bit result.

Table 9: Formats for Scalar Cryptography SHA Instructions

Instruction	BS			FUNC5					RS2					RS1					FUNC3			RD					OPCODE						
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sha256sig0	0	0	0	1	0	0	0	0	0	0	1	0						0	0	1						0	0	1	0	0	1	1
sha256sig1	0	0	0	1	0	0	0	0	0	0	1	1						0	0	1						0	0	1	0	0	1	1
sha256sum0	0	0	0	1	0	0	0	0	0	0	0	0						0	0	1						0	0	1	0	0	1	1
sha256sum1	0	0	0	1	0	0	0	0	0	0	0	1						0	0	1						0	0	1	0	0	1	1

2.3.2.2.1 SHA256SIG0 and SHA256SIG1

Instructions sha256sig0 and sha256sig1 perform the Sigma 0 and Sigma 1 functions used in the block decomposition step of the SHA256 hash function, defined in Section 2.2.1 Both

instructions operate on the data stored in RS1, and store the result in RD. The assembly implementations of these two instructions follow syntax in Figure 12.

```
sha256sig0 rd, rs1
sha256sig1 rd, rs1
```

Figure 12: RISC-V SHA256 sig0 and sig1 Instructions Assembly Syntax

The pseudocode in Figure 13 summarizes the process of both instructions [6].

```
function clause execute (SHA256SIG0(rs1,rd)) = {
       let inb : bits(32) = X(rs1)[31..0];
       let result : bits(32) = ror32(inb, 7) ^ ror32(inb, 18) ^ (inb >> 3);
       X(rd) = EXTS(result);
       RETIRE_SUCCESS
}

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

Figure 13: SAIL Pseudocode for RISC-V SHA256 sig0 and sig1 Instructions

The assembly code in Figure 14 describes a standard use case for the sha256sig0 and sha256sig1 instructions, used in generating a new double-word for the message schedule. Registers a0, a1, a2, and a3 hold previous double-words W_{i-16}, W_{i-15}, W_{i-7}, and W_{i-2} respectively, and sp holds the address of message schedule double-word W_0. Adjusting the memory offsets for the loads and stores, this code could be repeated forty-eight times for generation of the entire message schedule.

```
add             t0, a0, a2      //add w[i-16], w[i-7]
sha256sig0      t1, a1          //sig0 w[i-15]
add             t0, t0, t1      //add to total
sha256sig1      t1, a3          //sig1 w[i-2]
add             t0, t0, t1      //new double-word
sw              t0, 64(sp)      //store new double-word
lw              a0, 4(sp)       //load w[i-16]
lw              a1, 8(sp)       //load w[i-15]
lw              a2, 40(sp)      //load w[i-7]
lw              a3, 60(sp)      //load w[i-2]
```

Figure 14: RISC-V SHA256 sig0 and sig1 Instructions Assembly Use Case

The reason the scalar instructions targets only two functions of the message scheduling process rather than whole double-word generation is because each new double-word requires the four previous, which are too many inputs for a scalar instruction. With the new vector instructions, generation of whole double-words (or multiple) can be targeted with a single instruction by using longer register widths.

2.3.2.2.2 SHA256SUM0 and SHA256SUM1

Instructions sha256sum0 and sha256sum1 perform the Sum 0 and Sum 1 functions used in the hashing rounds of the SHA256 hash function, defined in Section 2.2.1. Both instructions operate on the data stored in RS1, and store the result in RD. The assembly implementations of these two instructions follow the syntax in Figure 15.

```
sha256sum0 rd, rs1
sha256sum1 rd, rs1
```

Figure 15: RISC-V SHA256 sum0 and sum1 Instructions Assembly Syntax

The pseudocode in Figure 16 summarizes the process of both instructions [6].

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

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

Figure 16: SAIL Pseudocode for RISC-V SHA256 sum0 and sum1 Instructions

The assembly code in Figure 17 describes a standard use case for the sha256sum0 and sha256sum1 instructions, used in completing a single SHA256 hash round. In this example, a0 and a1 initially hold the round constant and message schedule double-word, and registers s0-s7

hold previous round hash values of H_1^i through H_8^i. Calculating intermediate values a-h and adding them to the previous hash values have been combined to reduce instructions.

```
and           t0, s4, s5     //calculate T1
xori          t1, s4, -1
and           t1, t1, s6
xor           t1, t1, t0     //ch function
sha256sum1    t0, s4
add           t1, t1, s7     //add h
add           t1, t1, t0     //add sum1
add           t1, t1, a0     //add round constant
add           t1, t1, a1     //add message schedule

sha256sum0    t2, s0         //calculate T2
and           t0, s0, s1
and           t3, s0, s2
and           t4, s1, s2
xor           t0, t0, t3
xor           t0, t0, t4     //maj function
add           t2, t2, t0     //add sum0

add           s7, s7, s6     //calculate a-h and add to H0-H7
add           s6, s6, s5
add           s5, s5, s4
add           a4, s3, t1
add           s4, s4, a4
add           s3, s3, s2
add           s2, s2, s1
add           s1, s1, s0
add           a0, t2, t1
add           s0, s0, a0
```

Figure 17: RISC-V SHA256 sum0 and sum1 Instructions Assembly Use Case

The scalar instructions again targeted only two functions in a broader hash round due to lack of input space. Vector instructions instead can be used to complete the entire hash round, rather than just the sum0 and sum1 calculations. The multiple inputs required (hash, message schedule, and round key) make this challenging even for vector instructions, but the payoff will be worth it given how many scalar instructions the hash round takes to complete.

2.4 Existing Vector Cryptography Instructions

Existing cryptography instructions are examined from Intel's ISA to determine what new instructions would be feasible to add to RISC-V. Intel's publications detail use of AES and SHA instructions that utilize Intel's 128-bit XMM SIMD registers available in some processors.

2.4.1 Intel AES Instructions

Intel's AES instructions were introduced with the 2010 Intel Core processor family [8]. Four instructions handle AES middle and final round encryption and decryption (AESENC, AESENCLAST, AESDEC, AESDECLAST), similar to the scalar RISC-V aes32esmi, aes32esi, aes32dsmi, and aes32dsi instructions respectively, but targeting completion of full rounds instead of single bytes. Two other instructions (AESKEYGENASSIST and AESIMC) support key schedule generation [8]. The encryption and decryption instructions will be examined. The key schedule will not be examined due to a potential RISC-V vector AES scheduling instruction having little overall effect on encryption/decryption runtime, as explained in Section 2.1.1.

The four encryption and decryption instructions take in the entire 128-bit input block from register xmm1, and the entire 128-bit round key from xmm2/m128. The instructions are then capable of completing the full Substitute Bytes, Shift Rows, Mix Columns, and Add Round Key steps as needed in a single instruction (or the inverse operations for decrypt). Intel's provided pseudocode in Figure 18 details these operations [8].

```
AESENC xmm1, xmm2/m128
Tmp := xmm1
Round Key := xmm2/m128
Tmp := ShiftRows (Tmp)
Tmp := SubBytes (Tmp)
Tmp := MixColumns (Tmp)
xmm1 := Tmp xor Round Key

AESDEC xmm1, xmm2/m128
Tmp := xmm1
Round Key := xmm2/m128
Tmp := InvShift Rows (Tmp)
Tmp := InvSubBytes (Tmp)
Tmp := InvMixColumns (Tmp)
xmm1 := Tmp xor Round Key
```

```
AESENCLAST xmm1, xmm2/m128
Tmp := xmm1
Round Key := xmm2/m128
Tmp := Shift Rows (Tmp)
Tmp := SubBytes (Tmp)

xmm1 := Tmp xor Round Key

AESDECLAST xmm1, xmm2/m128
State := xmm1
Round Key := xmm2/m128
Tmp := InvShift Rows (State)
Tmp := InvSubBytes (Tmp)

xmm1:= Tmp xor Round Key
```

Figure 18: Pseudocode for Intel's AES Instructions

Assembly code in Figure 19 shows how Intel's AES instructions can be used to complete a full AES-128 encryption [8]. The input is first added (XOR'ed) to round key 0, before one AES instruction is completed for each of the ten requisite encryption rounds. Xmm15, holding the data being encrypted, stays constant for every round, while the second operand iterates though the round keys in xmm0-10.

```
; AES-128 encryption sequence.
; The data block is in xmm15.
; Registers xmm0-xmm10 hold the round keys(from 0 to 10 in this order).
; In the end, xmm15 holds the encryption result.
pxor xmm15, xmm0              ; Whitening step (Round 0)
aesenc xmm15, xmm1            ; Round 1
aesenc xmm15, xmm2            ; Round 2
aesenc xmm15, xmm3            ; Round 3
aesenc xmm15, xmm4            ; Round 4
aesenc xmm15, xmm5            ; Round 5
aesenc xmm15, xmm6            ; Round 6
aesenc xmm15, xmm7            ; Round 7
aesenc xmm15, xmm8            ; Round 8
aesenc xmm15, xmm9            ; Round 9
aesenclast xmm15, xmm10       ; Round 10
```

Figure 19: Intel's AES Instructions Assembly Use Case

This format of AES encryption instruction would also be desirable for RISC-V. Given proper vector register widths, inputs and round keys could be taken as 128-bit operands each round, reducing the number of required instructions for a single round to one, down from the RISC-V Cryptography extension's sixteen.

2.4.2 Intel SHA256 Instructions

Intel's Fast SHA256 Implementation instructions were introduced with the 2011 family of Intel Core Processors [7]. The specification features one SHA256 algorithm defined in four ways depending on instruction extensions present: SSE processors without AVX, processors with AVX1, processors with AVX2 and the RORX2 instruction, and processors with AVX2 and the RORX8 instruction [7].

Intel notes that the hashing rounds are inherently serial, depending directly on the hash values H_1^{i-1} through H_8^{i-1} to compute the next set, and thus cannot be parallelized and should likely just be calculated with integer instructions. Message scheduling can be parallelized in two ways: generating one message at a time for multiple parallel input blocks, or generating multiple words for a single input block. Because of the many circumstances in which there may not be enough parallel input blocks to fill out the entire 128-bit registers, Intel recommends parallelizing the message scheduling for a single input instead [7].

Per the message scheduling description detailed in Section 2.2, the first sixteen double-words $w[0]$ through $w[15]$ are derived from splitting of the 512-bit input. The next forty-eight double-words use the message scheduling calculations in equation 15.

Because the furthest reference in the calculation of $w[i]$ is $w[i-16]$, Intel loads only the last sixteen double-words into XMM registers, shown in Table 10 [7].

Table 10: Intel's Register Placement of SHA256 Message Schedule

Register	Double-word			
X3	w[i-1]	w[i-2]	w[i-3]	w[i-4]
X2	w[i-5]	w[i-6]	w[i-7]	w[i-8]
X1	w[i-9]	w[i-10]	w[i-11]	w[i-12]
X0	w[i-13]	w[i-14]	w[i-15]	w[i-16]

Using these sixteen double-words, $w[i-16]$, $w[i-7]$, and $s0[i]$ can be calculated for the next four double-words $w[i], w[i+1], w[i+2]$ and $w[i+3]$. However, because calculation of $s1[i+2]$ and $s1[i+3]$ require double-words $w[i]$ and $w[i+1]$, which have not been calculated yet, $s1[i]$ must be computed in pairs. This process allows two sets of two double-words to be calculated from four XMM registers, resulting in a 128-bit output to be used in calculation of the next four double-words of the message schedule [7].

All four SHA256 algorithm types follow this same basic loop for message scheduling, with the only difference being that the RORX2 variation can compute message schedules for two parallel input blocks, and RORX8 can compute for eight parallel input blocks [7].

RISC-V could likely run a similar instruction for message scheduling, computing multiple double-words per instruction to speed up the calculation. The lack of a hashing function in this paper is surprising, despite the serial nature, as it could likely still speed up the process by nature of the larger registers. A RISC-V instruction for completing, at the very least, as single hashing

round should still be investigated. Instructions that calculate Sum 0 and Sum 1 in parallel would be a backup instruction that could still help to streamline multiple parallel hashes.

2.5 Vicuna Vector Coprocessor

Implementation on an existing fully functional RISC-V processor is needed to provide a proof-of-concept for new vector cryptography instructions. Due to the newness of the vector extension, few open-source cores support the instructions. Vicuna is one of the only RISC-V vector coprocessor cores available online, and is fully synthesizable in System Verilog [10].

The Vicuna RISC-V Vector Coprocessor implements the Zve32x Suite of the RISC-V Vector extension. This suite lacks floating-point and ternary arithmetic support and has a max element width of 32 bits. The processor works in tandem with the Ibex core, an RV32IM processor also fully synthesizable in System Verilog [10].

As the only current point of hardware reference, all new vector cryptography instructions proposed by this thesis will target the 32-bit architecture of the Vicuna and Ibex processors. Vicuna uses default vector register width VMUL of 128 bits, and an ALU internal bus width ALUOP of 64 bits. Both VMUL and ALUOP are configurable by the user, but ALUOP must be less than or equal to one half VMUL, and both must be powers of two. Because the ALU bus width is half of VMUL, data in vector registers is also processed in halves, requiring at least two clock cycles in each pipeline stage for every instruction. Vicuna has no forwarding, branch prediction, out-of-order operation, and other optimizations in favor of timing predictability and demonstration of worst-case execution time [10].

Memory files in .vmem format with support for RISC-V vector instructions up to revision 0.8 are compiled using the GNU toolchain for RISC-V, and can be run on Vicuna [10].

Chapter 3

ANALYSIS

3.1 Methods

Recommendations for new vector cryptography AES and SHA instructions for RISC-V are informed by the outlined goals of RISC-V, the current scalar RISC-V cryptography instructions, the current RISC-V vector instructions, and the reasoning behind Intel's vector cryptography instructions.

From the goals outlined in the RISC-V Manual and RISC-V Scalar Cryptography Extension, proposed instructions aim to prioritize performance, though various possible hardware solutions are considered. Instructions avoid defining hardware where possible. Instruction recommendations target only existing, internationally recognized standards AES and SHA, and not attempt to guess at or support future theoretical standards [2][6].

The RISC-V Vector extension was referred to for instruction formats. Instructions utilize the existing OP-V major opcode, and utilize the same bit placement of func6, func3, RS1, RS2, and RD fields where possible to maintain standardization across instructions and uphold the ease of implementation RISC-V is grounded in [16].

The Scalar Cryptography extension was referenced for functions to implement. Vector versions of existing scalar AES and SHA instructions were given the most priority. SM3 and SM4 were not adapted to vector formats due to their limited global use. Instructions for encryption standards not referenced in the scalar specification were not considered. Vector cryptography instruction viability was judged by the number of instructions necessary for a complete encryption/hash versus the number of scalar instructions necessary to perform the same task.

Intel's vector cryptography implementations of SHA and AES were referenced for ways to translate the scalar cryptography instructions to vector equivalents. The limitations and benefits of certain function implementations Intel described in their ISA documents were considered in the creation of the new RISC-V vector cryptography instructions [7][8].

A modified Vicuna vector coprocessor was used to observe real hardware implementations of proposed instructions. The processor was used to identify potential strengths

and weaknesses of potential vector cryptography instructions, including the need for baseline register widths and ALU bus sizes for certain instruction implementations. An instruction implementation and execution on the Vicuna processor was used to prove that the new instructions worked as predicted [10].

3.2 RISC-V Vector AES Instructions

Because all versions of AES use 128-bit input blocks and round keys, vector AES instructions require vector lengths VLEN of at least 128 bits for maximum performance. By taking in the entire state array and round key as inputs, AES encryption instructions can likely perform full encryption rounds, as Intel's instructions do.

Vector AES instructions on RISC-V are all of the integer type, as the inputs, outputs, and calculations of AES do not use floating-point or masked operands. Vector AES instructions should utilize the existing vector arithmetic opcode of {1010111}, as the tables on pages 95 and 96 of the RISC-V Vector Extension show that twenty-two new integer instructions can still fit under this opcode [16]. Using an existing opcode would streamline implementation and require fewer lookup tables for decoding the instructions.

Both the input state and round keys of AES require 128 bits, so vector-vector is the only operand combination allowed. All vector AES instructions therefore have func3 values of {000} to denote integer vector-vector type.

Unused func6 values listed by the tables on pages 95 and 96 of the RISC-V Vector Extension define which instruction is being called in memory [16]. Values will begin at {110010} for the new vector cryptography instructions. This is the first unused func6 value that features enough following unused func6 values to group all new vector cryptography instructions together.

Two inputs VS1 and VS2 are required for the input state array and round key, and an output register VD is needed for the encrypted block.

Together, these requirements grant the four instruction formats in Table 11 for new vector instructions vaesds, vaesdsm, vaeses, and vaesesm. The moniker ".vv" is used in the assembly syntax of RISC-V vector instructions to denote the input type – in this case, vector-vector.

Table 11: Formats for Vector Cryptography AES Instructions

Instruction	FUNC6						VM	VS2					VS1					FUNC3			VD				OPCODE						
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
vaesds.w	1	1	0	0	1	0												0	0	0						1	0	1	0	1	1	1
vaesdsm.w	1	1	0	0	1	1												0	0	0						1	0	1	0	1	1	1
vaeses.w	1	1	0	1	0	0												0	0	0						1	0	1	0	1	1	1
vaesesm.w	1	1	0	1	0	1												0	0	0						1	0	1	0	1	1	1

3.2.1 VAESESM and VAESES

Like the scalar instructions, full-round vector AES encrypt instructions require both middle and end round instructions to account for the lack of Mix Columns in the last round. VS2 holds the state array, and VS1 holds the round key. VD is the destination register for the round result. To follow the assembly implementations of previous RISC-V instructions, these two instructions follow the syntax in Figure 20.

```
vaesesm.vv vd, vs2, vs1, vm
vaeses.vv, vd, vs2, vs1, vm
```

Figure 20: RISC-V Vector AES Encrypt Instructions Assembly Syntax

Both instructions take the 128-bit state array input from VS2 and perform the Shift Rows step. The Rijndael S-Box substitution is performed on all 16 bytes of the shifted input. The substituted array then undergoes a full Mix Column (this step is skipped during final round instruction vaeses) and is XOR'ed with the round key in VS1. The 128-bit result is stored in VD to serve as input to the next round. Pseudocode provided in Figure 21, mimicking the RISC-V Scalar Cryptography extension SAIL pseudocode examples, summarizes the operations of both instructions. Functions shiftrows_fwd, sbox_fwd, and mixcolumn_fwd are performed exactly as described in Section 2.1.2.

```
function clause execute (VAESESM(vs2, vs1, vd)) = {
       let sr : bits(128) = aes_128_shiftrows_fwd(X(vs2)[127..0]);
       let sb : bits(128) = aes_128_sbox_fwd(sr[127..0]);
       let ms : bits(128) = aes_128_mixcolumn_fwd(sb[127..0]);
       X(vd) = sb[127..0] ^ X(vs1)[127..0];
       RETIRE_SUCCESS
}

function clause execute (VAESES(vs2, vs1, vd)) = {
       let sr : bits(128) = aes_128_shiftrows_fwd(X(vs2)[127..0]);
       let sb : bits(128) = aes_128_sbox_fwd(sr[127..0]);
       X(vd) = sb[127..0] ^ X(vs1)[127..0];
       RETIRE_SUCCESS
}
```

Figure 21: SAIL Pseudocode for RISC-V Vector AES Encrypt Instructions

The assembly code in Figure 22 defines a full AES-128 encryption using the new vector instructions. Vector registers v1-v11 hold the round keys, and vector v0 holds the state array input. Upon code completion, v0 holds the encrypted output.

```
vxor.vv       v0, v0, v1     //add round key
vaesesm.vv    v0, v0, v2     //round 1
vaesesm.vv    v0, v0, v3     //round 2
vaesesm.vv    v0, v0, v4     //round 3
vaesesm.vv    v0, v0, v5     //round 4
vaesesm.vv    v0, v0, v6     //round 5
vaesesm.vv    v0, v0, v7     //round 6
vaesesm.vv    v0, v0, v8     //round 7
vaesesm.vv    v0, v0, v9     //round 8
vaesesm.vv    v0, v0, v10    //round 9
vaeses.vv     v0, v0, v11    //round 10
```

Figure 22: RISC-V Vector AES Encrypt Instructions Assembly Use Case

3.2.2 VAESDSM and VAESDS

The full-round vector AES decrypt instructions also require both middle and end round instructions. VS2 holds the state array, and VS1 holds the round key. VD is the destination register for the round result. To follow the assembly implementations of previous RISC-V instructions, these two instructions follow the syntax in Figure 23.

```
vaesdsm.vv vd, vs2, vs1, vm

vaesds.vv, vd, vs2, vs1, vm
```

Figure 23: RISC-V Vector AES Decrypt Instructions Assembly Syntax

As before, the instructions execute the inverse rounds of the encryption instructions. Both instructions take the 128-bit state array input from VS2 and perform the Inverse Shift Rows step. The Inverse Rijndael S-Box substitution is performed on all sixteen bytes of the shifted input. The substituted array is XOR'ed with the round key in VS1, then undergoes a full Inverse Mix Column (this step is skipped during final round instruction vaesds). The 128-bit result is stored in VD to serve as input to the next round. Pseudocode provided in Figure 24, mimicking the RISC-V Scalar Cryptography extension SAIL pseudocode examples, summarizes the operations of both instructions. Functions shiftrows_inv, sbox_inv, and mixcolumn_inv are performed exactly as described in Section 2.1.3.

```
function clause execute (VAESDSM(vs2, vs1, vd)) = {
       let sr : bits(128) = aes_128_shiftrows_inv(X(vs2)[127..0]);
       let sb : bits(128) = aes_128_sbox_inv(sr[127..0]);
       let rk : bits(128) = sb[127..0] ^ X(vs1)[127..0];;
       X(vd) = aes_128_mixcolumn_inv(rk[127..0]);
       RETIRE_SUCCESS
}

function clause execute (VAESDS(vs2, vs1, vd)) = {
       let sr : bits(128) = aes_128_shiftrows_inv(X(vs2)[127..0]);
       let sb : bits(128) = aes_128_sbox_inv(sr[127..0]);
       X(vd) = sb[127..0] ^ X(vs1)[127..0];
       RETIRE_SUCCESS
}
```

Figure 24: SAIL Pseudocode for RISC-V Vector AES Decrypt Instructions

The assembly code in Figure 25 defines a full AES-128 decryption using the new vector instructions. Vector registers v1-v11 hold the round keys, and vector v0 holds the state array input. Upon code completion, v0 holds the decrypted output.

```
vxor.vv        v0, v0, v11    //add round key
vaesdsm.vv     v0, v0, v10    //round 1
vaesdsm.vv     v0, v0, v9     //round 2
vaesdsm.vv     v0, v0, v8     //round 3
vaesdsm.vv     v0, v0, v7     //round 4
vaesdsm.vv     v0, v0, v6     //round 5
vaesdsm.vv     v0, v0, v5     //round 6
vaesdsm.vv     v0, v0, v4     //round 7
vaesdsm.vv     v0, v0, v3     //round 8
vaesdsm.vv     v0, v0, v2     //round 9
vaesds.vv      v0, v0, v1     //round 10
```

Figure 25: RISC-V Vector AES Decrypt Instructions Assembly Use Case

3.3 RISC-V Vector SHA256 Instructions

As stated by Intel in their cryptography instruction publications, the SHA256 hashing rounds are largely serial [7]. Because they can't be parallelized, the whole hash must be completed in a single function. Registers need to be 256 bits in length to store the operands and result.

Message scheduling could be implemented exactly as in the Intel publication, generating four new double-words W_i through W_{i+3} per round form the previous sixteen [7]. This mandates at least 256-bit vector registers to hold all sixteen previous double-words as two inputs. In order to fill the entire 256-bit destination register, the new RISC-V message scheduling instruction will instead generate eight double-words instead of four.

Both instructions are integer type with vector-vector operands, as the inputs, outputs, and operations of SHA256 do not use floating-point or masked operands. This is not defined explicitly by func3 as it is in other RISC-V vector instructions to allow room for double-word selection in the hash instruction, which will be discussed in Section 3.3.2. To properly identify the function of bits {14:12}, they are renamed DWS (double-word select) for the SHA instructions.

Both RISC-V vector SHA instructions use the OP-V major opcode {1010111} for reasons defined in Section 3.2. Unused func6 values, beginning where the Vector AES instructions left off in Section 3.2 at {110110}, define which instruction is being called in memory.

The purpose of VS2, VS1, and VD change based on the instruction due to the different number of operands for message schedule generation and hashing. All three registers are defined per-instruction in sections 3.3.1 and 3.3.2. All registers have to be at least 256-bits in length.

Together, these requirements grant the two instruction formats in Table 12 for new vector instructions vsha256hash and vsha256sch. The moniker ".vv" is used in the assembly syntax of RISC-V vector instructions to denote the input type.

Table 12: Formats for Vector Cryptography SHA256 Instructions

Instruction	FUNC6						VM	VS2					VS1					DWS			VD				OPCODE						
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
vsha256sch.w	1	1	0	1	1	0												0	0	0						1	0	1	0	1	1	1
vsha256hash.w	1	1	0	1	1	1																				1	0	1	0	1	1	1

3.3.1 VSHA256SCH

The vsha256sch instruction takes the previous sixteen double-words of the SHA 256 message schedule and generates the next eight. To follow the assembly implementations of previous RISC-V instructions, this instruction follows the syntax in Figure 26.

```
vsha256sch.vv vd, vs2, vs1, vm
```

Figure 26: RISC-V Vector SHA256 Scheduling Instruction Assembly Syntax

This instruction requires a vector register width of at least 256 bits. VS1 holds double-words W_{i-1} through W_{i-8} of the message schedule, and VS2 holds the double-words W_{i-9}, through W_{i-16}. These double-words are used to generate the next eight 32-bit double words of the schedule. Sigma 1 and Sigma 0 as described in Section 2.2.1 are used to calculate the eight words, which are concatenated and stored in VD. Pseudocode provided in Figure 27, mimicking the RISC-V Scalar Cryptography extension SAIL pseudocode examples, summarizes the operation of the instruction.

```
function clause execute (VSHA256SCH(vs2, vs1, vd, dws)) = {
          let x0:  bits(128) = X(vs2)[127..0];
          let x1:  bits(128) = X(vs2)[255..128];
          let x2:  bits(128) = X(vs1)[127..0];
          let x3:  bits(128) = X(vs1)[255..128];
          let dw1: bits(32) = x0[31..0] + x2[63..32] + sig0(x0[63..32]) + sig1(x3[95..64]);
          let dw2: bits(32) = x0[63..32] + x2[95..64] + sig0(x0[95..64]) + sig1(x3[127..96]);
          let dw3: bits(32) = x0[95..64] + x2[127..96] + sig0(x0[127..96]) + sig1(dw1[31..0]);
          let dw4: bits(32) = x0[127..96] + x3[31..0] + sig0(x1[31..0]) + sig1(dw2[31..0]);
          let dw5: bits(32) = x1[31..0] + x3[63..32] + sig0(x1[63..32]) + sig1(dw3[31..0]);
          let dw6: bits(32) = x1[63..32] + x3[95..64] + sig0(x1[95..64]) + sig1(dw4[31..0]);
          let dw7: bits(32) = x1[95..64] + x3[127..96] + sig0(x1[127..96]) + sig1(dw5[31..0]);
          let dw8: bits(32) = x1[127..96] + dw1[31..0] + sig0(x2[31..0]) + sig1(dw6[31..0]);
          X(vd) = dw8 @ dw7 @ dw6 @ dw5 @ dw4 @ dw3 @ dw2 @ dw1;
          RETIRE_SUCCESS
}
```

Figure 27: SAIL Pseudocode for RISC-V Vector SHA256 Scheduling Instruction

The assembly code in Figure 28 defines a full SHA256 message schedule generation using the new vector instructions. Vector register v0 initially holds the double-words W_{i-9} through W_{i-16} and v1 initially holds double-words W_{i-1} through W_{i-8} The eight new double-words of the previous round are used to calculate the eight new double-words of the current round, granting the cyclical nature of the registers. Upon code completion, each register v0-v7 will hold eight double-words of the message schedule.

```
vsha256sch.vv  v2, v0, v1     //generate double-words 17-24
vsha256sch.vv  v3, v1, v2     //generate double-words 25-32
vsha256sch.vv  v4, v2, v3     //generate double-words 33-40
vsha256sch.vv  v5, v3, v4     //generate double-words 41-48
vsha256sch.vv  v6, v4, v5     //generate double-words 49-56
vsha256sch.vv  v7, v5, v6     //generate double-words 57-64
```

Figure 28: RISC-V Vector SHA256 Scheduling Instruction Assembly Use Case

3.3.2 VSHA256HASH

The vsha256hash instruction takes the current eight hash values, eight message schedule double-words, and eight round constants, and completes one whole SHA256 hashing round. This requires a ternary instruction, like the existing vector add-multiply instructions of the RISC-V Vector extension, using VS1, VS2, and VD as operands. To follow the assembly implementations of previous RISC-V instructions, this instruction follows the syntax in Figure 29.

```
vsha256hash.vv vd, vs2, vs1, dws, vm
```

Figure 29: RISC-V Vector SHA256 Hash Instruction Assembly Syntax

This instruction requires a vector register width of at least 256 bits and is destructive to VD. VD holds the current hashing values H_1^i through H_8^i. VS2 holds eight consecutive message schedule double-words, and VS1 holds eight consecutive round constants. Message schedule and round constant for the current round, W_i and K_i are selected from VS2 and VS1 with DWS. T1 and T2 are calculated with Sum0, Sum1, Ch, and Maj as described in Section 2.2.1, and intermediate values a, b, c, d, e, f, g and h are calculated for the current round. Finally, the intermediate values are added to the current hash values to create the next eight hash values, which are concatenated and stored in VD, overwriting the previous data. Pseudocode provided in Figure 30, mimicking the RISC-V Scalar Cryptography extension SAIL pseudocode examples, summarizes the operation of the instruction.

```
function clause execute (VSHA256HASH(vs2, vs1, vd, dws)) = {
          let w  : bits(32) = (X(vs2)[255..0] >> (dws*32))[31..0];
          let k  : bits(32) = (X(vs1)[255..0] >> (dws*32))[31..0];
          let t1 : bits(32) = X(vd)[31..0] + sig1(X(vd)[127..96]) + ch(X(vd)[127..32]) + k + w;
          let t2 : bits(32) = sig0(X(vd)[255:224]) + maj(X(vd)[255:160])
          let h  : bits(32) = X(vd)[63..32];
          let g  : bits(32) = X(vd)[95..64];
          let f  : bits(32) = X(vd)[127..96];
          let e  : bits(32) = X(vd)[159..128] + t1;
          let d  : bits(32) = X(vd)[191..160];
          let c  : bits(32) = X(vd)[223..192];
          let b  : bits(32) = X(vd)[255..224];
          let a  : bits(32) = t1 + t2;
          let ha : bits(256) = (X(vd)[255:224] + a) @ (X(vd)[223:192] + b)
                             @ (X(vd)[191:160] + c) @ (X(vd)[159:128] + d)
                             @ (X(vd)[127:96] + e) @ (X(vd)[95..64] + f)
                             @ (X(vd)[63:32] + g) @ (X(vd)[31..0] + h);
          X(vd) = ha;
          RETIRE_SUCCESS
}
```

Figure 30: SAIL Pseudocode for RISC-V Vector SHA256 Hash Instruction

The assembly code in Figure 31 defines eight SHA256 hashing rounds using the new vector instructions. This code can be repeated eight times for a complete hash. Vector register v1 initially holds round constants K_i through K_{i+7} and v2 initially holds the double-words W_i, through W_{i+7} from the message schedule. DWS, the value at the end of each instruction, selects the

double-word to pull from v1 and v2, corresponding to the current round. Register v0 holds the previous round's hashing values H_1^{i-1} through H_8^{i-1}, and is overridden with the current round's hashing values H_1^i through H_8^i. Because v1 and v2 can only hold eight double-words, they must be refreshed with the next set from memory every eight hashing rounds (or replaced with two other registers holding the new data). Upon code completion, v0 will hold the hashed output.

```
vsha256hash.vv v0, v2, v1, 0  //Hash round i
vsha256hash.vv v0, v2, v1, 1  //Hash round i+1
vsha256hash.vv v0, v2, v1, 2  //Hash round i+2
vsha256hash.vv v0, v2, v1, 3  //Hash round i+3
vsha256hash.vv v0, v2, v1, 4  //Hash round i+4
vsha256hash.vv v0, v2, v1, 5  //Hash round i+5
vsha256hash.vv v0, v2, v1, 6  //Hash round i+6
vsha256hash.vv v0, v2, v1, 7  //Hash round i+7
```

Figure 31: RISC-V Vector SHA256 Hash Instruction Assembly Use Case

3.4 Vivado Instruction Simulation on Vicuna

To provide a proof-of concept for the instruction design choices made in this thesis, a full vector AES-128 encryption and SHA256 message schedule was performed using a modified Vicuna vector coprocessor in Vivado. An AES-128 decryption was not performed, as the calculations for encryption and decryption are identical save for constants and shift directions. A full SHA256 hash was not integrated, as the use of ternary functions requires use of the MUL block of Vicuna, requiring a separate and more intensive integration than that of the AES encrypt and SHA message schedule, which can both be performed in the ALU.

Successful integration of the new instructions prove three key aspects of the proposed vector cryptography extension: that the vector arithmetic opcode {1010111} can still be used without overlap, that the unused func6 values listed in the RISC-V Vector Extension are freely usable, and that enough data can be received from memory to perform full encryption rounds in a single clock cycle on hardware. Furthermore, the integration proves the functionality and usefulness of the vaeses, vaesesm, and vsha256sch instructions, and by extension the vaesds and vaesdsm instructions. While vsha256hash was tested directly, it can be assumed that if the method of determining opcodes, func6, and register widths for the other instructions are all

operational, that integration of vsha256hash with the same design decisions would require little to no deviation from the format recommended in this thesis.

Vicuna uses a case statement in the decoder to interpret incoming func6 values when a vector opcode is detected. The type of vector opcode is then stored as a string in a nested struct that is passed through the coprocessor pipeline to the ALU [10]. Integration of each instruction requires adding vector cryptography instruction type definition strings to these structs so that they may be asserted when a vector cryptography instruction is identified, and so that the ALU can then perform the necessary functions [10]. RISC-V's example substitution box System Verilog implementation was utilized to avoid remaking the same large lookup tables, which is available on the RISC-V Scalar Cryptography extension Github page [6]. Strings ALU_VAESE and ALU_VSHA were added to the structs to identify an vector AES encrypt or vector SHA type instruction, and strings ALU_VAES_ESMI, ALU_VAES_ESI, and ALU_VSHA_SCH were added to further identify which specific instruction is begin called. Because the vector opcode, VS1, VS2, VD, func3, and vm bits all act the same, the func6 value is the only portion of the instruction that requires new decoding hardware.

The Vicuna ALU features 4 optional buffers, all of which are enabled by default: BUF_VREG, BUF_OPERANDS, BUF_INTERMEDIATE, and BUF_RESULTS [10]. Hardware for the calculation of the result of each new instruction is placed between the OPERANDS and INTERMEDIATE buffers, where most other ALU calculations including shifts and arithmetic are also conducted.

The vproc_decoder, vproc_alu, and vproc_pkg (which defines the structs passed form the decoder to ALU) are the only files of Vicuna that need alteration for the new instructions to work [10]. Code additions made to each of these files are provided in Appendix A for future duplication of results.

Simulated implementation utilization of the Vicuna processor updated with vector AES encrypt and SHA message scheduling indicated 43833 LUTs and 26064 flip-flops, up from 36717 LUTs and 25693 flip-flops on the base processor. The 19.4% increase in LUTs is likely due to the sixteen parallel lookup tables for byte substitution during the AES encryption stage. If a scheduler

could be used to individually send bytes through a single substitute block, LUT's could be reduced. The 1.44% increase in flip-flops is within the expected range for the logic added.

Vicuna does not feature forwarding, branch prediction, or other timing acceleration hardware, resulting in many read-after-write hazards causing stalls in the execution of the new instructions [10]. Execution time of the instructions is largely based on the hardware they are running on and the quality of the implementation, and so will not be considered in the following simulations. Vicuna and the associated Ibex processor also do not support the scalar cryptography extension, so no timing comparison between the two extensions can be performed. It will be left to future research to calculate the energy per operation and execution time of these instructions to determine their viability in processors where these parameters are of greater consideration.

3.4.1 Full Vector AES Encryption Hardware Simulation

A full vector AES encryption requires integration of the vaeses and vaesesm instructions outlined in Section 3.2.1. Data memory is initialized with the example given in Table 13 [17]. The expected output after both the first and the tenth encryption rounds are also listed for comparison to the simulated results.

Table 13: AES Example Input, Round Keys, and Outputs

Object	Data
Input	54 77 6F 20 4F 6E 65 20 4E 69 6E 65 20 54 77 6F
Encryption Key	54 68 61 74 73 20 6D 79 20 4B 75 6E 67 20 46 75
Round Key 1	E2 32 FC F1 91 12 91 88 B1 59 E4 E6 D6 79 A2 93
Round Key 2	56 08 20 07 C7 1A B1 8F 76 43 55 69 A0 3A F7 FA
Round Key 3	D2 60 0D E7 15 7A BC 68 63 39 E9 01 C3 03 1E FB
Round Key 4	A1 12 02 C9 B4 68 BE A1 D7 51 57 A0 14 52 49 5B
Round Key 5	B1 29 3B 33 05 41 85 92 D2 10 D2 32 C6 42 9B 69
Round Key 6	BD 3D C2 87 B8 7C 47 15 6A 6C 95 27 AC 2E 0E 4E
Round Key 7	CC 96 ED 16 74 EA AA 03 1E 86 3F 24 B2 A8 31 6A
Round Key 8	8E 51 EF 21 FA BB 45 22 E4 3D 7A 06 56 95 4B 6C
Round Key 9	BF E2 BF 90 45 59 FA B2 A1 64 80 B4 F7 F1 CB D8
Round Key 10	28 FD DE F8 6D A4 24 4A CC C0 A4 FE 3B 31 6F 26
Round 1 Output	58 47 08 8B 15 B6 1C BA 59 D4 E2 E8 CD 39 DF CE
Round 10 Output	29 C3 50 5F 57 14 20 F6 40 22 99 B3 1A 02 D7 3A

Assembly code was written and compiled with the RISCV GNU toolchain compiler. Dummy instructions were inserted where new vector instructions would be placed in memory, as the compiler cannot recognize and translate them. The vaeses and vaesesm instructions were then formatted in binary according to the instruction formats proposed in Section 3.2, and converted to hex before manually replacing the dummy instructions in the .vmem file produced by the GNU compiler. Full assembly code for an AES encryption using the new instructions is available in Appendix B.

The input and round keys were manually stored in column-major order in memory, each time starting with the last column of the state array and ending with the first. This is so that when they are loaded to the register file, the most-significant double-word of each register would store the first column, and so on, appearing as they do in Table 13. A vector register width of 256 bits and ALU width of 128 bits are used for the simulation.

Figure 32 depicts the timing diagram of a full AES encryption after appropriate hardware has been added to Vicuna. The top two values ALU_VAESE and ALU_VAES_ESMI/ESI are

derived from the instruction func6 value and indicate that Vicuna has recognized the vector AES instructions. This proves the func6 and opcode choices for these instructions work.

Due to Vicuna's ALU taking half a vector register at a time, each instruction first operates only on the filled 128-bit half of the vector registers VS1 and VS2 containing the input and round keys. The same instruction then performs the operation on the second half the of the vector registers, which are unused in this simulation and simply hold all 0's. Ten permutations of this half-half register calculation can be seen – one for each round – with stalls in between due to read-after-write hazards.

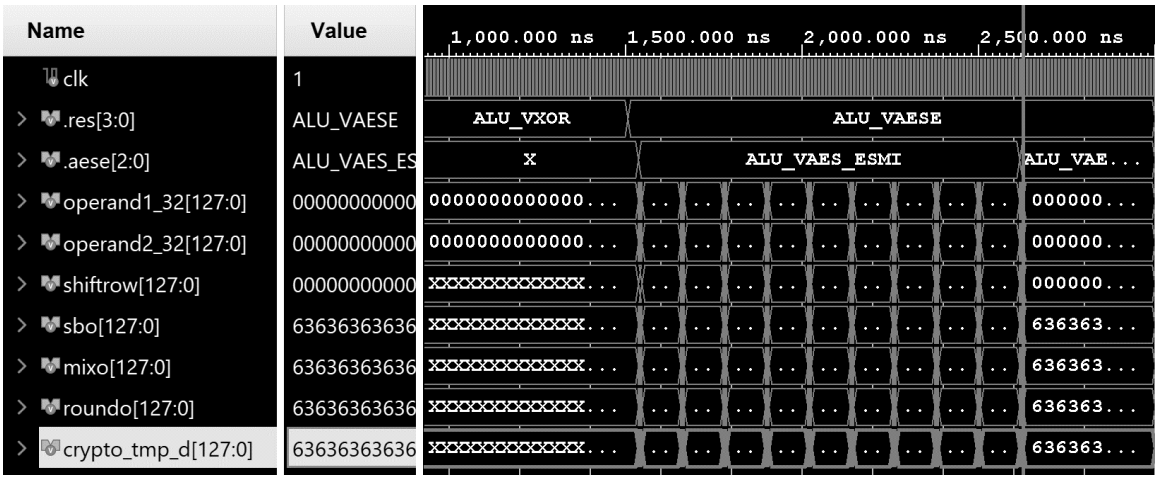

Figure 32: Vivado Wave Window Depiction of Vector AES-128 Encryption

Zooming in on the first round in Figure 33, operand2_32 holds the input data (already XOR'ed with the encryption key) and operand1_32 holds round key 1. Registers "shiftrow," "sbo," and "mixo" hold the outputs of the shift rows, sbox, and mix columns steps respectively. Register "roundo" holds which value is the input to the final roundkey XOR, and is used to skip the mix columns step in the last round. The output, held in "cryptography_tmp_d", adheres to the expected round 1 value from Table 13. From these signals, it is proven that vaesesm can be identified from memory, and that a 128-bit input and 128-bit round key can be used to complete a full encryption round in one clock cycle of Vicuna.

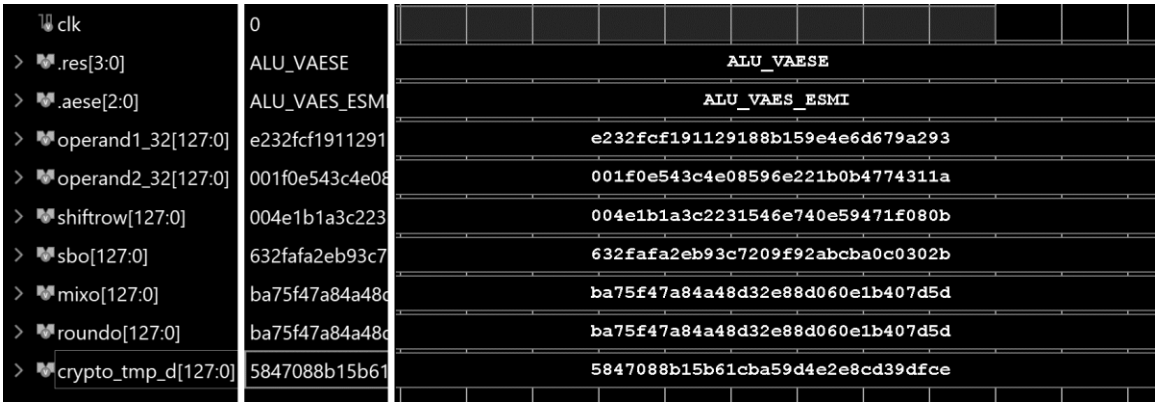

Figure 33: Vivado Wave Window Depiction of First AES Round

Zooming in on the last round in Figure 34, all signals behave the same, except “roundo” holds the sbox output instead of the mix columns output, as is required in the final round of AES encryption. This value is XOR’ed with the final round key and stored in “cryptography_tmp_d,” which matches our expected final round output in Table 13. Matching the expected output signals that a complete AES-128 encryption has been performed on the Vicuna coprocessor using the new vector instructions vaeses and vaesesm.

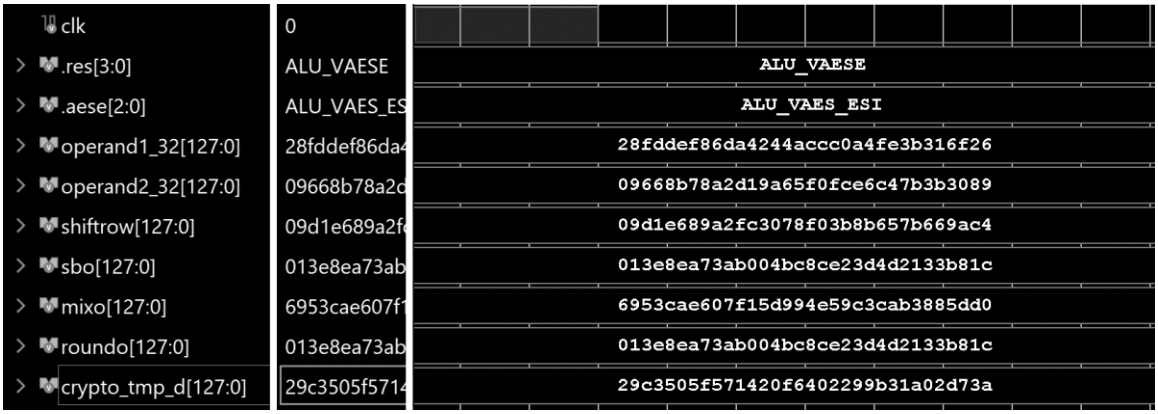

Figure 34: Vivado Wave Window Depiction of Last AES Round

Assuming no loads or stores, the new vector AES instructions have been proven capable of completing a full encryption round in a single instruction, compared to sixteen with the scalar cryptography extension – a 93.75% reduction in number of required instructions and memory

space. The difference in execution between vector and scalar instructions is summarized in Table 14.

Table 14: AES Encryption Round Vector vs. Scalar

Vector Assembly	Scalar Assembly

`vaesesm.vv v0, v0, v1 //round 1`	`aes32esmi a0, a0, t0, 0`
	`aes32esmi a0, a0, t1, 1`
	`aes32esmi a0, a0, t2, 2`
	`aes32esmi a0, a0, t3, 3`
	`aes32esmi a1, a1, t1, 0`
	`aes32esmi a1, a1, t2, 1`
	`aes32esmi a1, a1, t3, 2`
	`aes32esmi a1, a1, t0, 3`
	`aes32esmi a2, a2, t2, 0`
	`aes32esmi a2, a2, t3, 1`
	`aes32esmi a2, a2, t0, 2`
	`aes32esmi a2, a2, t1, 3`
	`aes32esmi a3, a3, t3, 0`
	`aes32esmi a3, a3, t0, 1`
	`aes32esmi a3, a3, t1, 2`
	`aes32esmi a3, a3, t2, 3 //round 1`

3.4.2 Full Vector SHA Message Schedule Hardware Simulation

A full vector SHA message schedule generation requires integration of the vsha256sch instruction outlined in Section 3.3.1. Data memory is initialized with the first two lines given in Table 15. This 256-bit initial message also serves as the first sixteen double-words of the message schedule per the message schedule calculation in Section 2.2.1. The expected first 8 generated words, and the expected last eight generated words, are listed for comparison to the simulated results [18]. The first eight words should be generated after a single vsha256sch instruction, and the last eight will be generated after five subsequent scheduling instructions.

Table 15: SHA256 Example Input, First 8 Double-Words, and Last 8 Double-Words

706F7274	736D6F75	74688000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000050
2FC59E5F	A1F54B8F	F158037D	0C9E5996	01ED1087	E7FAE04A	2A531BDF	5BDB7019
.....	.....	.....	.....	.....	.....	.....	.....
.....	.....	.....	.....	.....	.....	.....	.....
.....	.....	.....	.....	.....	.....	.....	.....
.....	.....	.....	.....	.....	.....	.....	.....
8091D44B	60B9CC0F	FABEF733	ECD282D4	3045E214	F62E8215	F27D9B62	54A70545

As with the AES simulation, assembly code was written and compiled with the RISCV GNU toolchain compiler. Dummy instructions were inserted where new vector instructions would be placed in memory, as the compiler cannot recognize and translate them. The vsha256sch instruction was then formatted in binary according to the instruction format proposed in Section 3.3, and converted to hex before manually replacing the dummy instructions in the .vmem file produced by the GNU compiler. Full assembly code for a SHA256 message schedule using the new instruction is available in Appendix B.

The input message, denoted as the first two rows of Table 15, were written manually in memory starting with the first double-word {706F7274} and ending with the last double-word {00000050}. This is organized such that the first double-word will be loaded into the least-significant double-word of register VS2, and the last double-word will be in the most-significant double-word of VS1. These represent double-words W_{i-16} through W_{i-1} in the calculation of the first 8 new double-words of the message schedule. These eight new double-words are then used with the previous eight to calculate the next eight, and so on until all sixty-four double-words are created. A vector register width of 512 bits and ALU width of 256 bits was used for the simulation.

Figure 35 depicts the timing diagram of a full SHA256 message schedule generation after appropriate hardware has been added to Vicuna. The top two values ALU_SHA and ALU_SHA_SCH are derived from the instruction func6 value and indicate that Vicuna has

recognized the vector SHA instructions. This proves the func6 and opcode choices for this instruction work.

Due to Vicuna's ALU taking half a vector register at a time, each instruction first operates only on the filled 256-bit half of the vector registers containing the previous double-words. The same instruction then performs the operation on the second half the of the vector registers, which are unused in this simulation and simply hold all 0's. Six uses of the vsha256sch instruction can be seen, with stalls between for read-after-write hazards and a pause after three have completed to store the calculated values before they are overridden.

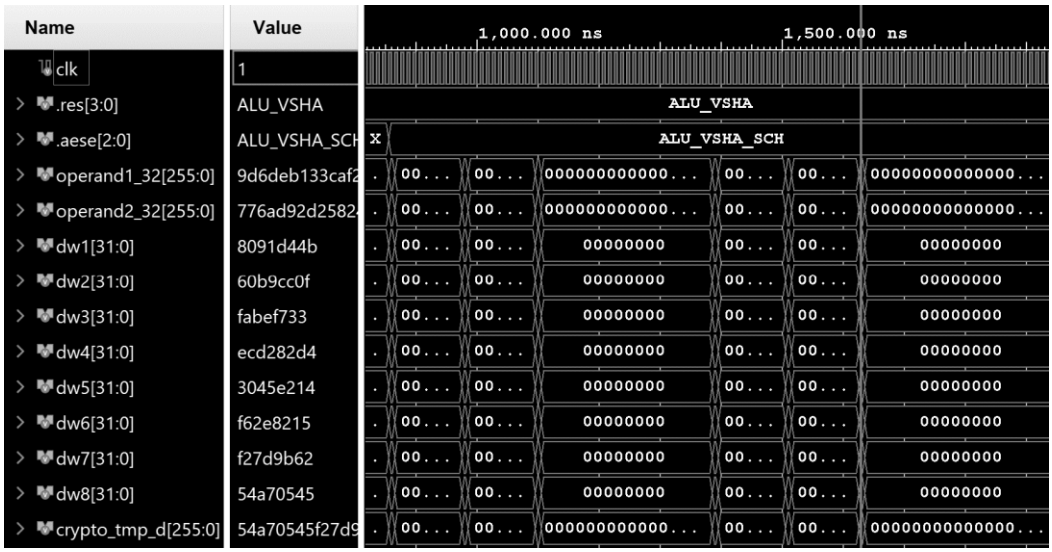

Figure 35: Vivado Wave Window Depiction of Vector SHA256 Message Schedule Generation

Zooming in on the first instance of vasha256sch in Figure 36, "operand2_32" holds the first eight double-words and "operand1_32" holds the second eight. These are used to calculate "dw1" through "dw8," the next set of eight double-words. These double-words match the third row W_{17} through W_{24} in Table 15 exactly, indicating a successful operation. Register "cryptography_tmp_d" then holds the concatenated output for use in the next calculation.

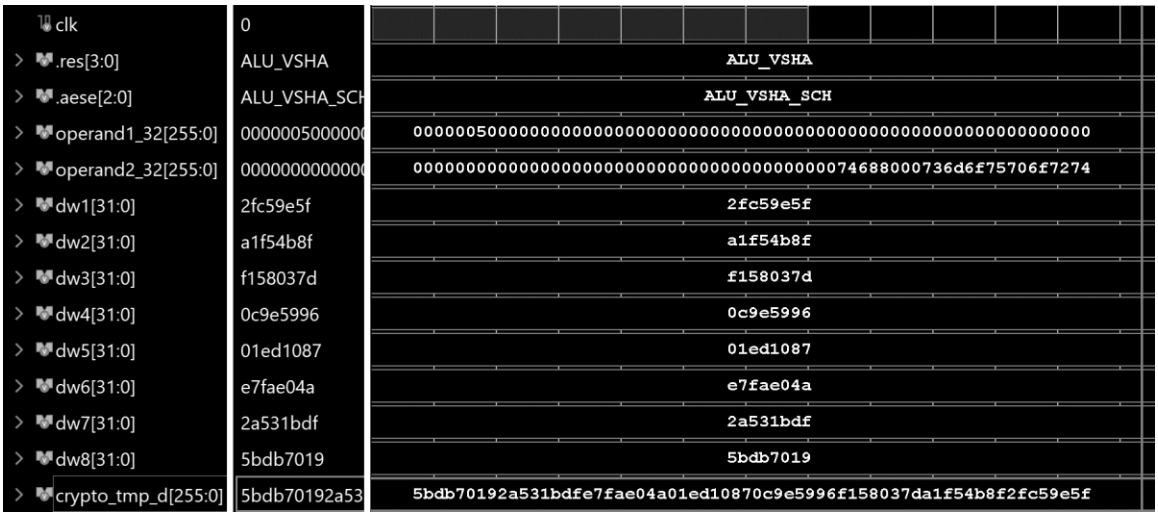

Figure 36: Vivado Wave Window Depiction of First 8 SHA256 Double-Words

Looking at the last instance of vsha256sch in Figure 37 shows the last eight calculated double-words. These also match up with the last eight double-words W_{57} through W_{64} in Table 15, proving that this instruction, as well as the previous four instances between the first and last, were all successful. Six instructions generating eight double-words each equates to forty-eight calculated double-words, that in addition to the original sixteen make up the entire 64-entry message schedule.

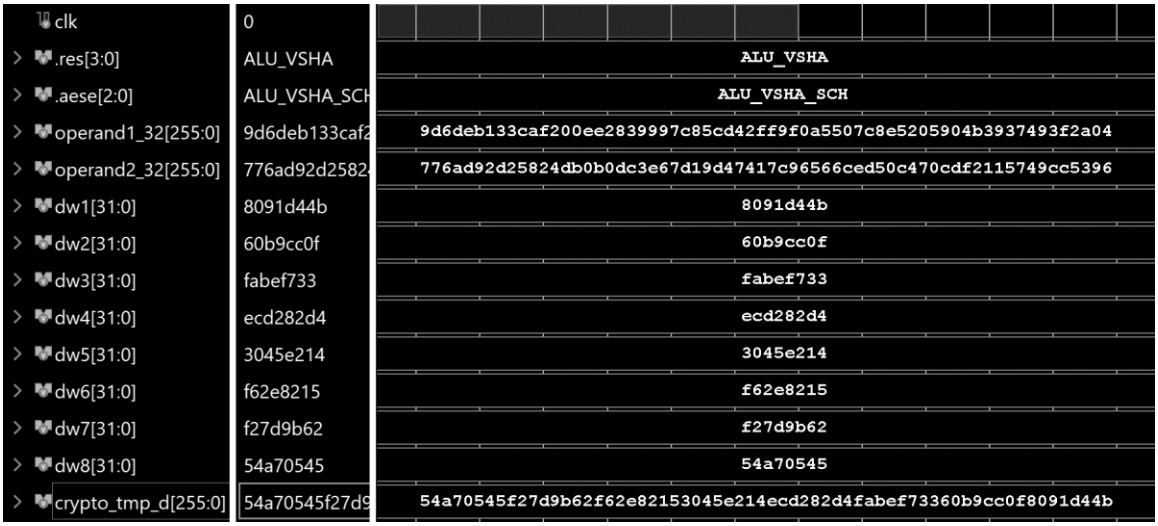

Figure 37: Vivado Wave Window Depiction of Last 8 SHA256 Double-Words

With proof vsha256sch works, eight double-words can be computed with a single instruction with the proposed Vector Cryptography extension. Assuming no loads or stores, this would require 40 instructions at minimum with the Scalar Cryptography extension – a 97.5% decrease in required instructions and memory space. The difference in execution between vector and scalar instructions is summarized in Table 16.

Table 16: Generation of 8 SHA256 Message Schedule Double-Words Vector vs. Scalar

Vector Assembly		Scalar Assembly	
vsha256sch.vv	v2, v0, v1 //8 new DWs	add	t0, a0, a2
		sha256sig0	t1, a1
		add	t0, t0, t1
		sha256sig1	t1, a3
		add	a4, t0, t1 //1 new DW
		.	
		.	
		.	
		x8	

Chapter 4

CONCLUSION

Vector instructions are viewed as a way to improve cryptography algorithms by both the RISC-V Foundation and some of its commercial competitors. Vector instructions can easily handle the large bit-widths of cryptography operands and are capable of completing a number of encryption steps in parallel. AES and SHA2 represent two prolific cryptography standards supported by the original RISC-V Cryptography Extension that stand to benefit from vector instructions. Proposed instructions vaeses and vaesesm would be capable of completing whole AES encryption rounds in a single instruction with inputs of the state array and the round key if vector registers are longer than 128 bits, and inverted decryption rounds would be completed in the same way with vaesdsm and vaesds. The SHA256 message schedule can be generated 8 double-words per vsha256sch instruction with the last 16 double-words as input and 256-bit registers. Whole SHA256 hashing rounds can also be completed with the hash values, current message schedule double-word and current round constant as inputs in a single ternary instruction vsha256hash with 256-bit registers. Successful implementation of vaeses, vaesesm, and vsha256sch in a RISC-V vector coprocessor Vicuna displays the ability of the architecture to complete whole encryption steps with large operands in single clock cycles, proves the suggested instruction formats are readable with no overlap or collisions with existing instructions, and highlights the potential of vector cryptography instructions in RISC-V by significantly reducing the memory space and coding time required to perform cryptography operations. Vector AES encryption rounds complete with 93.75% fewer instructions than the scalar equivalent, and vector SHA256 message schedule generation completes with 97.5% fewer instructions than the scalar equivalent, assuming no loads or stores. The hardware cost in this analysis was a 19.4% increase in LUTs and 1.44% increase in flip-flops on the edited Vicuna processor.

These instructions provide only a suggested starting point for the upcoming Vector Cryptography Extension, and future work must aim at adapting instructions for SM3, SM4, and bit-manipulation functions into the vector format. Additionally, while the success of the vector AES instructions can be construed to the AES decryption instructions due to nearly identical execution,

implementation and testing of the vsha256hash instruction would be necessary before it could be seriously considered for addition to an official extension.

REFERENCES

[1] "History of RISC-V," *RISCV.org*. Accessed: Jun. 04, 2022. [Online]. Available: https://riscv.org/about/history/

[2] A. Waterman and K. Asanovic, Eds., *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*. University of California, Berkeley: CS Division, EECS Department, Dec. 13, 2019. Accessed: Jun. 05, 2022. [Online]. Available: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[3] K. McMahon, "RISC-V International Ratifies 15 New Specifications, Opening Up New Possibilities for RISC-V Designs," *RISCV.org*, Dec. 2, 2021. Accessed: Jun. 05, 2022. [Online]. Available: https://riscv.org/announcements/2021/12/riscv-ratifies-15-new-specifications/

[4] *Announcing the ADVANCED ENCRYPTION STANDARD (AES)*. Federal Information Processing Standards, 2001. Accessed: Jun. 05, 2022. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

[5] W. May, *FIPS PUB 180-4 FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION Secure Hash Standard (SHS)*. Gaithersburg, MD: National Institute of Standards and Technology, Aug. 2015. Accessed: Jun. 05, 2022. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

[6] *RISC-V Cryptography Extensions Volume I*. Switzerland: RISC-V Foundation, Feb. 18, 2022. Accessed: Jun. 04, 2022. [Online]. Available: https://github.com/riscv/riscv-crypto

[7] J. Guilford, K. Yap, and V. Gopal, "Fast SHA-256 Implementations on Intel® Architecture Processors," 2012. Accessed: Jun. 05, 2022. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/sha-256-implementations-paper.pdf

[8] S. Gueron, "Intel® Advanced Encryption Standard (AES) New Instructions Set," 2010. Accessed: Jun. 05, 2022. [Online]. Available: https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf

[9] “Cryptographic Standards and Guidelines,” *CSRC.NIST.gov*, Aug. 2021. Accessed: Jun. 05, 2022. [Online]. Available: https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines

[10] M. Platzer and P. Puschner, “Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation,” in *33rd Euromicro Conference on Real-Time Systems*, 2021, vol. 196. Accessed: Jun. 05, 2022. [Online]. Available: https://github.com/vproc/vicuna

[11] R. Rimkiene, “What is AES Encryption and How Does It Work?,” *CyberNews*, Dec. 11, 2020. Accessed: Jun. 05, 2022. [Online]. Available: https://cybernews.com/resources/what-is-aes-encryption/

[12] A. Kak, “Lecture 8: AES: The Advanced Encryption Standard Lecture Notes on ‘Computer and Network Security,’” Purdue Univeristy, Feb. 03, 2022. Accessed: Jun. 04, 2022. [Online]. Available: https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture8.pdf

[13] “Rijndael S-box,” *Wikipedia*, May 2020. Accessed: Jun. 05, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Rijndael_S-box

[14] “What is SHA Encryption? SHA-1 vs SHA-2,” *sectigo.com*, July 07, 2021. Accessed: Jun. 05, 2022. [Online]. Available: https://sectigo.com/resource-library/what-is-sha-encryption

[15] C. Bellet, “Hashing with SHA-256,” *Medium.com*, Mar. 14, 2018. Accessed: Jun. 05, 2022. [Online]. Available: https://medium.com/biffures/part-5-hashing-with-sha-256-4c2afc191c40

[16] *RISC-V “V” Vector Extension*. Switzerland: RISC-V Foundation, Sept. 20, 2021. Accessed: Jun. 04, 2022. [Online]. Available: https://github.com/riscv/riscv-v-spec

[17] “AES Example - Input (128 bit key and message).” Accessed: Jun. 05, 2022. [Online]. Available: https://www.kavaliro.com/wp-content/uploads/2014/03/AES.pdf

[18] T. Chitty, “The Mathematics of Bitcoin — SHA256,” *The Startup*, May 25, 2020. Accessed: Jun. 05, 2022. [Online]. Available: https://medium.com/swlh/the-mathematics-of-bitcoin-74ebf6cefbb0

APPENDICES

APPENDIX A

Verilog Additions to Vicuna

vproc_pkg.sv

```
typedef enum logic [2:0] {
    ALU_VAES_ESI,
    ALU_VAES_ESMI,
    ALU_VSHA_SCH,
    ALU_VSHA_HASH
} opcode_alu_aese;

typedef enum logic [3:0] {
    ALU_VADD,
    ALU_VAADD,
    ALU_VAND,
    ALU_VOR,
    ALU_VXOR,
    ALU_VSHIFT,
    ALU_VSEL,
    ALU_VSELN,
    ALU_VAESE,
    ALU_VAESD,
    ALU_VSHA
} opcode_alu_res;
```

vproc_decoder.sv

```
{6'b110100, 3'b000}: begin  // vaesei
    unit_o              = UNIT_ALU;
    mode_o.alu.opx2.res = ALU_VAESE;
    mode_o.alu.opx1.aese = ALU_VAES_ESI;
    mode_o.alu.inv_op1  = 1'b0;
    mode_o.alu.inv_op2  = 1'b0;
    mode_o.alu.op_mask  = ALU_MASK_NONE;
    mode_o.alu.cmp      = 1'b0;
    mode_o.alu.masked   = instr_masked;
end


{6'b110101, 3'b000}: begin  // vaesemi
    unit_o              = UNIT_ALU;
    mode_o.alu.opx2.res = ALU_VAESE;
    mode_o.alu.opx1.aese = ALU_VAES_ESMI;
    mode_o.alu.inv_op1  = 1'b0;
    mode_o.alu.inv_op2  = 1'b0;
    mode_o.alu.op_mask  = ALU_MASK_NONE;
    mode_o.alu.cmp      = 1'b0;
    mode_o.alu.masked   = instr_masked;
end


{6'b110110, 3'b000}: begin  // vshasch
    unit_o              = UNIT_ALU;
    mode_o.alu.opx2.res = ALU_VSHA;
    mode_o.alu.opx1.aese = ALU_VSHA_SCH;
    mode_o.alu.inv_op1  = 1'b0;
    mode_o.alu.inv_op2  = 1'b0;
    mode_o.alu.op_mask  = ALU_MASK_NONE;
    mode_o.alu.cmp      = 1'b0;
    mode_o.alu.masked   = instr_masked;
end
```

vproc_alu.sv

```
//crypto
    logic [ALU_OP_W    -1:0] crypto_tmp_q,     crypto_tmp_d;    //crypto output
    logic [ALU_OP_W-1:0] shiftrow;  //shift rows output
    logic [ALU_OP_W-1:0] sbo;       //sbox output
    logic [ALU_OP_W-1:0] mixo;      //mix columns output
    logic [ALU_OP_W-1:0] roundo;    //holds mixo or sbo to XOR'ed with round key
    logic aes;              //enables AES sbox
    logic sm4;              //enables SM4 sbox for future integration
    logic dec;              //enables inverse AES sbox
    logic [7:0] s0;         //first column byte
    logic [7:0] s1;         //second column byte
    logic [7:0] s2;         //third column byte
    logic [7:0] s3;         //fourth column byte

    logic [127:0] x0;
    logic [127:0] x1;
    logic [127:0] x2;
    logic [127:0] x3;
    logic [31:0] dw1;       //first generated SHA256 double-word
    logic [31:0] dw2;       //second generated SHA256 double-word
    logic [31:0] dw3;       //third generated SHA256 double-word
    logic [31:0] dw4;       //fourth generated SHA256 double-word
    logic [31:0] dw5;       //fifth generated SHA256 double-word
    logic [31:0] dw6;       //sixth generated SHA256 double-word
    logic [31:0] dw7;       //seventh generated SHA256 double-word
    logic [31:0] dw8;       //eighth generated SHA256 double-word
    logic [255:0] shao;     //concatonated output of 8 double-words


    `define ROR(a,b) ((a >> b) | (a << 32-b))
    `define SRL(a,b) ((a >> b)             )
    `define Sig0(a) `ROR(a, 7)^`ROR(a,18)^`SRL(a, 3)
    `define Sig1(a) `ROR(a, 17)^`ROR(a,19)^`SRL(a, 10)


    always_comb begin
        case (state_ex1_q.mode.opx2.res)
        ALU_VAESE: begin

            //shift rows
            shiftrow      = {operand2_32[127:120], operand2_32[87:80],
                             operand2_32[47:40], operand2_32[7:0],
                             operand2_32[95:88], operand2_32[55:48],
                             operand2_32[15:8], operand2_32[103:96],
                             operand2_32[63:56], operand2_32[23:16],
                             operand2_32[111:104], operand2_32[71:64],
                             operand2_32[31:24], operand2_32[119:112],
                             operand2_32[79:72], operand2_32[39:32]};

            case(state_ex1_q.mode.opx1.aese)
                ALU_VAES_ESMI: begin
                    aes=1;
                    sm4=0;
                    dec=0;
                    roundo=mixo;
                end
                ALU_VAES_ESI: begin
                    aes=1;
                    sm4=0;
                    dec=0;
                    roundo=sbo;
            end
            endcase

            crypto_tmp_d=  roundo^operand1_32;
        end
        ALU_VSHA: begin
            aes=0;
            sm4=0;
```

```
            dec=0;
            crypto_tmp_d=shao;
        end
        default: begin
            aes=0;
            sm4=0;
            dec=0;
        end
        endcase
    end

//SHA256 message schedule calculation. Commented out for AES test
    always_comb begin
        x0=operand2_32[127:0];
        x1=operand2_32[255:128];
        x2=operand1_32[127:0];
        x3=operand1_32[255:128];

        dw1=x0[31:0]+x2[63:32]+(`Sig0(x0[63:32]))+(`Sig1(x3[95:64]));
        dw2=x0[63:32]+x2[95:64]+(`Sig0(x0[95:64]))+(`Sig1(x3[127:96]));
        dw3=x0[95:64]+x2[127:96]+(`Sig0(x0[127:96]))+(`Sig1(dw1[31:0]));
        dw4=x0[127:96]+x3[31:0]+(`Sig0(x1[31:0]))+(`Sig1(dw2[31:0]));
        dw5=x1[31:0]+x3[63:32]+(`Sig0(x1[63:32]))+(`Sig1(dw3[31:0]));
        dw6=x1[63:32]+x3[95:64]+(`Sig0(x1[95:64]))+(`Sig1(dw4[31:0]));
        dw7=x1[95:64]+x3[127:96]+(`Sig0(x1[127:96]))+(`Sig1(dw5[31:0]));
        dw8=x1[127:96]+dw1[31:0]+(`Sig0(x2[31:0]))+(`Sig1(dw6[31:0]));
        shao = {dw8, dw7, dw6, dw5, dw4, dw3, dw2, dw1};
    end

//AES encryption calculation

    //sub bytes
    sbox #(
        .ALU_OP_W (ALU_OP_W)   // ALU operand width in bits
    ) sboxtransform(
        .si(shiftrow),
        .aes(aes),
        .sm4(sm4),
        .dec(dec),
        .so (sbo)
    );

    //mix columns
    always_comb begin
        for (int i = 1; i <= ALU_OP_W / 32; i++) begin
            s0=sbo[(32 * i)-8 +: 8];
            s1=sbo[(32 * i)-16 +: 8];
            s2=sbo[(32 * i)-24 +: 8];
            s3=sbo[(32 * i)-32 +: 8];

            mixo[(32 * i)-8 +: 8]= ((s0<<1)^(s0[7] ? 8'h1b : 8'h0))^(s1^((s1<<1)^
                                                    (s1[7] ? 8'h1b : 8'h0)))^s2^s3;
            mixo[(32 * i)-16 +: 8]= s0^((s1<<1)^(s1[7] ? 8'h1b : 8'h0))^(s2^((s2<<1)^
                                                    (s2[7] ? 8'h1b : 8'h0)))^s3;
            mixo[(32 * i)-24 +: 8]=s0^s1^((s2<<1)^(s2[7] ? 8'h1b : 8'h0))^(s3^((s3<<1)^
                                                    (s3[7] ? 8'h1b : 8'h0)));
            mixo[(32 * i)-32 +: 8]=(s0^((s0<<1)^(s0[7] ? 8'h1b : 8'h0)))^s1^s2^((s3<<1)^
                                                    (s3[7] ? 8'h1b : 8'h0));
        end
    end
```

sbox.sv

```
module sbox #(
        parameter int unsigned          ALU_OP_W         = 64   // ALU operand width in
bits
    )(
        input logic [ALU_OP_W-1:0] si,
        input logic aes,
```

```
    input logic sm4,
    input logic dec,
    output logic [ALU_OP_W-1:0] so
);

//the riscv_crypto_aes_sm4_sbox is taken from the rtl folder
//on the official RISC-V Scalar Cryptography Github [6]

riscv_crypto_aes_sm4_sbox riscv_sbox1 (
    .aes (aes),
    .sm4 (sm4),
    .dec (dec),
    .in (si[7:0]),
    .out (so[7:0])
);

riscv_crypto_aes_sm4_sbox riscv_sbox2 (
    .aes (aes),
    .sm4 (sm4),
    .dec (dec),
    .in (si[15:8]),
    .out (so[15:8])
);

riscv_crypto_aes_sm4_sbox riscv_sbox3 (
    .aes (aes),
    .sm4 (sm4),
    .dec (dec),
    .in (si[23:16]),
    .out (so[23:16])
);

riscv_crypto_aes_sm4_sbox riscv_sbox4 (
    .aes (aes),
    .sm4 (sm4),
    .dec (dec),
    .in (si[31:24]),
    .out (so[31:24])
);

riscv_crypto_aes_sm4_sbox riscv_sbox5 (
    .aes (aes),
    .sm4 (sm4),
    .dec (dec),
    .in (si[39:32]),
    .out (so[39:32])
);

riscv_crypto_aes_sm4_sbox riscv_sbox6 (
    .aes (aes),
    .sm4 (sm4),
    .dec (dec),
    .in (si[47:40]),
    .out (so[47:40])
);

riscv_crypto_aes_sm4_sbox riscv_sbox7 (
    .aes (aes),
    .sm4 (sm4),
    .dec (dec),
    .in (si[55:48]),
    .out (so[55:48])
);

riscv_crypto_aes_sm4_sbox riscv_sbox8 (
    .aes (aes),
    .sm4 (sm4),
    .dec (dec),
    .in (si[63:56]),
    .out (so[63:56])
);
```

```
    riscv_crypto_aes_sm4_sbox riscv_sbox9 (
        .aes (aes),
        .sm4 (sm4),
        .dec (dec),
        .in (si[71:64]),
        .out (so[71:64])
    );

    riscv_crypto_aes_sm4_sbox riscv_sbox10 (
        .aes (aes),
        .sm4 (sm4),
        .dec (dec),
        .in (si[79:72]),
        .out (so[79:72])
    );

    riscv_crypto_aes_sm4_sbox riscv_sbox11 (
        .aes (aes),
        .sm4 (sm4),
        .dec (dec),
        .in (si[87:80]),
        .out (so[87:80])
    );

    riscv_crypto_aes_sm4_sbox riscv_sbox12 (
        .aes (aes),
        .sm4 (sm4),
        .dec (dec),
        .in (si[95:88]),
        .out (so[95:88])
    );

    riscv_crypto_aes_sm4_sbox riscv_sbox13 (
        .aes (aes),
        .sm4 (sm4),
        .dec (dec),
        .in (si[103:96]),
        .out (so[103:96])
    );

    riscv_crypto_aes_sm4_sbox riscv_sbox14 (
        .aes (aes),
        .sm4 (sm4),
        .dec (dec),
        .in (si[111:104]),
        .out (so[111:104])
    );

    riscv_crypto_aes_sm4_sbox riscv_sbox15 (
        .aes (aes),
        .sm4 (sm4),
        .dec (dec),
        .in (si[119:112]),
        .out (so[119:112])
    );

    riscv_crypto_aes_sm4_sbox riscv_sbox16 (
        .aes (aes),
        .sm4 (sm4),
        .dec (dec),
        .in (si[127:120]),
        .out (so[127:120])
    );

endmodule
```

APPENDIX B

Assembly Code

AES-128 Encryption

```
    .text
    .global main
main:
    la              sp, vdata_start
    li              t0, 4
    vsetvli         t0, t0, e32,m1,tu,mu

    vle32.v         v0, (sp)          //load data and round keys
    addi            sp, sp, 16
    vle32.v         v1, (sp)
    addi            sp, sp, 16
    vle32.v         v2, (sp)
    addi            sp, sp, 16
    vle32.v         v3, (sp)
    addi            sp, sp, 16
    vle32.v         v4, (sp)
    addi            sp, sp, 16
    vle32.v         v5, (sp)
    addi            sp, sp, 16
    vle32.v         v6, (sp)
    addi            sp, sp, 16
    vle32.v         v7, (sp)
    addi            sp, sp, 16
    vle32.v         v8, (sp)
    addi            sp, sp, 16
    vle32.v         v9, (sp)
    addi            sp, sp, 16
    vle32.v         v10, (sp)
    addi            sp, sp, 16
    vle32.v         v11, (sp)

    vxor.vv           v0, v0, v1     //add round key
    vaesesm.vv        v0, v0, v2     //round 1
    vaesesm.vv        v0, v0, v3     //round 2
    vaesesm.vv        v0, v0, v4     //round 3
    vaesesm.vv        v0, v0, v5     //round 4
    vaesesm.vv        v0, v0, v6     //round 5
    vaesesm.vv        v0, v0, v7     //round 6
    vaesesm.vv        v0, v0, v8     //round 7
    vaesesm.vv        v0, v0, v9     //round 8
    vaesesm.vv        v0, v0, v10    //round 9
    vaeses.vv         v0, v0, v11    //round 10


    .data
    .align 10
    .global vdata_start
vdata_start:
    .word           0x2054776F       //Data
    .word           0x4E696E65
    .word           0x4F6E6520
    .word           0x54776F20

    .word           0x67204675       //Round key 0
    .word           0x204B756E
    .word           0x73206D79
    .word           0x54686174

    .word           0xD679A293       //Round key 1
    .word           0xB159E4E6
    .word           0x91129188
    .word           0xE232FCF1
```

```
.word           0x00000000        //Round key 2
.word           0x00000000
.word           0x00000000
.word           0x00000050

.word           0xC3031EFB        //Round key 3
.word           0x6339E901
.word           0x157ABC68
.word           0xD2600DE7

.word           0x1452495B        //Round key 4
.word           0xD75157A0
.word           0xB468BEA1
.word           0xA11202C9

.word           0xC6429B69        //Round key 5
.word           0xD210D232
.word           0x05418592
.word           0xB1293B33

.word           0xAC2E0E4E        //Round key 6
.word           0x6A6C9527
.word           0xB87C4715
.word           0xBD3DC2B7

.word           0xB2A8316A        //Round key 7
.word           0x1E863F24
.word           0x74EAAA03
.word           0xCC96ED16

.word           0x56954B6C        //Round key 8
.word           0xE43D7A06
.word           0xFABB4522
.word           0x8E51EF21

.word           0xF7F1CBD8        //Round key 9
.word           0xA16480B4
.word           0x4559FAB2
.word           0xBFE2BF90

.word           0x3B316F26        //Round key 10
.word           0xCCC0A4FE
.word           0x6DA4244A
.word           0x28FDDEF8
```

SHA256 Message Schedule Generation

```
    .text
    .global main
main:
    la              a0, vdata_start
    li              t0, 8
    vsetvli         t0, t0, e32,m1,tu,mu
    vle32.v         v2, (a0)
    addi            a1, a0, 32
    vle32.v         v1, (a1)

    vsha256sch.vv      v0, v2, v1     //generate double-words 17-24
    vsha256sch.vv      v2, v1, v0     //generate double-words 25-32
    vsha256sch.vv      v1, v0, v2     //generate double-words 33-40

    addi            a1, a1, 32
    vse32.v         v0, (a1)
    addi            a1, a1, 32
    vse32.v         v1, (a1)
    addi            a1, a1, 32
    vse32.v         v2, (a1)

    vsha256sch.vv      v0, v2, v1     //generate double-words 17-24
    vsha256sch.vv      v2, v1, v0     //generate double-words 25-32
    vsha256sch.vv      v1, v0, v2     //generate double-words 33-40

    addi            a1, a1, 32
    vse32.v         v0, (a1)
    addi            a1, a1, 32
    vse32.v         v1, (a1)
    addi            a1, a1, 32
    vse32.v         v2, (a1)


    .data
    .align 10
    .global vdata_start
vdata_start:
    .word           0x706F7274       //DW 0-3
    .word           0x736D6F75
    .word           0x74688000
    .word           0x00000000

    .word           0x00000000       //DW 4-7
    .word           0x00000000
    .word           0x00000000
    .word           0x00000000

    .word           0x00000000       //DW 8-11
    .word           0x00000000
    .word           0x00000000
    .word           0x00000000

    .word           0x00000000       //DW 12-15
    .word           0x00000000
    .word           0x00000000
    .word           0x00000050
```