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

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>vii</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
</tbody>
</table>

### 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
# LIST OF TABLES

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

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. AES Key Schedule Generation Workflow</td>
<td>8</td>
</tr>
<tr>
<td>2. AES Encryption Workflow</td>
<td>9</td>
</tr>
<tr>
<td>3. AES Decryption Workflow</td>
<td>12</td>
</tr>
<tr>
<td>4. SHA256 Message Decomposition and Padding</td>
<td>15</td>
</tr>
<tr>
<td>5. SHA256 Hashing Workflow</td>
<td>16</td>
</tr>
<tr>
<td>6. RISC-V AES Encrypt Instructions Assembly Syntax</td>
<td>23</td>
</tr>
<tr>
<td>7. SAIL Pseudocode for RISC-V AES Encrypt Instructions</td>
<td>24</td>
</tr>
<tr>
<td>8. RISC-V AES Encrypt Instructions Assembly Use Case</td>
<td>24</td>
</tr>
<tr>
<td>9. RISC-V AES Decrypt Instructions Assembly Syntax</td>
<td>25</td>
</tr>
<tr>
<td>10. SAIL Pseudocode for RISC-V AES Decrypt Instructions</td>
<td>26</td>
</tr>
<tr>
<td>11. RISC-V AES Decrypt Instructions Assembly Use Case</td>
<td>26</td>
</tr>
<tr>
<td>12. RISC-V SHA256 sig0 and sig1 Instructions Assembly Syntax</td>
<td>28</td>
</tr>
<tr>
<td>13. SAIL Pseudocode for RISC-V SHA256 sig0 and sig1 Instructions</td>
<td>28</td>
</tr>
<tr>
<td>14. RISC-V SHA256 sig0 and sig1 Instructions Assembly Use Case</td>
<td>28</td>
</tr>
<tr>
<td>15. RISC-V SHA256 sum0 and sum1 Instructions Assembly Syntax</td>
<td>29</td>
</tr>
<tr>
<td>16. SAIL Pseudocode for RISC-V SHA256 sum0 and sum1 Instructions</td>
<td>29</td>
</tr>
<tr>
<td>17. RISC-V SHA256 sum0 and sum1 Instructions Assembly Use Case</td>
<td>30</td>
</tr>
<tr>
<td>18. Pseudocode for Intel’s AES Instructions</td>
<td>31</td>
</tr>
<tr>
<td>19. Intel’s AES Instructions Assembly Use Case</td>
<td>32</td>
</tr>
<tr>
<td>20. RISC-V Vector AES Encrypt Instructions Assembly Syntax</td>
<td>37</td>
</tr>
<tr>
<td>21. SAIL Pseudocode for RISC-V Vector AES Encrypt Instructions</td>
<td>38</td>
</tr>
<tr>
<td>22. RISC-V Vector AES Encrypt Instructions Assembly Use Case</td>
<td>38</td>
</tr>
<tr>
<td>23. RISC-V Vector AES Decrypt Instructions Assembly Syntax</td>
<td>38</td>
</tr>
<tr>
<td>24. SAIL Pseudocode for RISC-V Vector AES Decrypt Instructions</td>
<td>39</td>
</tr>
</tbody>
</table>
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 turn 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: vaesem, vaeses, vaesds, vaesds, sha256sch, and sha256hash. Descriptions, instruction formats, pseudocode, and suggested assembly use of all six are provided. A Verilog implementation of vaesem, vaeses, and vsha256sch follow, demonstrating a complete AES encryption and SHA256 message schedule generation in hardware. Instructions vaesds and vaesds are not implemented due to their computational similarity to vaeses and vaesem, and vsha256hash is left out due to its architectural complexity when compared to the other instructions.
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>
<thead>
<tr>
<th>AES Version</th>
<th>Encryption Key Length</th>
<th>Input Data Length</th>
<th>Number of Key Expansion Rounds</th>
<th>Number of Encryption/Decryption Rounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES-128</td>
<td>128</td>
<td>128</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>AES-192</td>
<td>192</td>
<td>128</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>AES-256</td>
<td>256</td>
<td>128</td>
<td>7</td>
<td>14</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>AES Version</th>
<th>Encryption Key Length</th>
<th>Number of Key Expansion Rounds</th>
<th>Double-Words Generated per Expansion Round</th>
<th>Total Generated Double-Words</th>
<th>Round Keys Created</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES-128</td>
<td>128</td>
<td>10</td>
<td>4</td>
<td>40</td>
<td>10</td>
</tr>
<tr>
<td>AES-192</td>
<td>192</td>
<td>8</td>
<td>6</td>
<td>48</td>
<td>12</td>
</tr>
<tr>
<td>AES-256</td>
<td>256</td>
<td>7</td>
<td>8</td>
<td>56</td>
<td>14</td>
</tr>
</tbody>
</table>
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].

\[
w_{i+4} = w_i \oplus g(w_{i+3}) \tag{1}
\]

where:
\[
g(X) = Sbox(ROL(X, 8)) \oplus \{RC_i, 0x00, 0x00, 0x00\}
\]

and:
\[
RC_1 = 0x01
\]
\[
RC_i = 0x02 \times RC_{i-1}
\]

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].

\[
w_{i+5} = w_{i+4} \oplus w_{i+1} \tag{2}
\]
\[
w_{i+6} = w_{i+5} \oplus w_{i+2} \tag{3}
\]
\[
w_{i+7} = w_{i+6} \oplus w_{i+4} \tag{4}
\]
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](image)

*Not in final round
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].

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

(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

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

\[
\begin{bmatrix}
S_{0.0} & S_{0.1} & S_{0.2} & S_{0.3} \\
S_{1.0} & S_{1.1} & S_{1.2} & S_{1.3} \\
S_{2.0} & S_{2.1} & S_{2.2} & S_{2.3} \\
S_{3.0} & S_{3.1} & S_{3.2} & S_{3.3}
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
S_{0.0} & S_{0.1} & S_{0.2} & S_{0.3} \\
S_{1.1} & S_{1.2} & S_{1.3} & S_{1.0} \\
S_{2.2} & S_{2.3} & S_{2.0} & S_{2.1} \\
S_{3.3} & S_{3.0} & S_{3.1} & S_{3.2}
\end{bmatrix}
\] (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.

\[
\begin{bmatrix}
2 & 3 & 1 & 1 \\
1 & 2 & 3 & 1 \\
1 & 1 & 2 & 3 \\
3 & 1 & 1 & 2
\end{bmatrix}
\times
\begin{bmatrix}
S_{0.0} & S_{0.1} & S_{0.2} & S_{0.3} \\
S_{1.1} & S_{1.2} & S_{1.3} & S_{1.0} \\
S_{2.2} & S_{2.3} & S_{2.0} & S_{2.1} \\
S_{3.3} & S_{3.0} & S_{3.1} & S_{3.2}
\end{bmatrix}
= 
\begin{bmatrix}
S'_{0.0} & S'_{0.1} & S'_{0.2} & S'_{0.3} \\
S'_{1.1} & S'_{1.2} & S'_{1.3} & S'_{1.0} \\
S'_{2.0} & S'_{2.1} & S'_{2.2} & S'_{2.3} \\
S'_{3.0} & S'_{3.1} & S'_{3.2} & S'_{3.3}
\end{bmatrix}
\] (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].

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].

\[
\begin{bmatrix}
S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\
S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\
S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\
S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3}
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\
S_{1,3} & S_{1,0} & S_{1,1} & S_{1,2} \\
S_{2,2} & S_{2,3} & S_{2,0} & S_{2,1} \\
S_{3,1} & S_{3,2} & S_{3,3} & S_{3,0}
\end{bmatrix}
\] (8)
Inverse Substitute Bytes uses the Inverse Rijndael S-Box in Table 4 [13].

**Table 4: AES Inverse Substitution Lookup Table**

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

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

\[
\begin{bmatrix}
  e & b & d & 9 \\
 9 & e & b & d \\
 d & 9 & e & b \\
 b & d & 9 & e
\end{bmatrix}
\times
\begin{bmatrix}
  s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\
  s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\
  s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\
  s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3}
\end{bmatrix}
= 
\begin{bmatrix}
  s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\
  s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\
  s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\
  s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3}
\end{bmatrix}
\tag{9}
\]

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].
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 $H_i^0$ 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. $H_i^0$ 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

<table>
<thead>
<tr>
<th>$K_0$</th>
<th>$K_1$</th>
<th>$K_2$</th>
<th>$K_3$</th>
<th>$K_4$</th>
<th>$K_5$</th>
<th>$K_6$</th>
<th>$K_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>428a2f90</td>
<td>71374491</td>
<td>b5c0fbcf</td>
<td>e9b5dbb5</td>
<td>3ef56258</td>
<td>87715cbb</td>
<td>a6f84e20</td>
<td>40e4164f</td>
</tr>
<tr>
<td>d807aa98</td>
<td>12835b01</td>
<td>243185be</td>
<td>550c7d33</td>
<td>72be5d74</td>
<td>80deb1fe</td>
<td>9bdc06a7</td>
<td>0353657c</td>
</tr>
<tr>
<td>e4b89c35</td>
<td>cefbe4786</td>
<td>0fc19de6</td>
<td>240ca1cc</td>
<td>2de92c6f</td>
<td>4a7484aa</td>
<td>5cb6a9d7</td>
<td>76f988da</td>
</tr>
<tr>
<td>93e5152</td>
<td>a831c66d</td>
<td>b00327c8</td>
<td>bf597f7c</td>
<td>c6e00fb3</td>
<td>d5a79147</td>
<td>06ca6351</td>
<td>14292db7</td>
</tr>
<tr>
<td>27b70a85</td>
<td>2e1b2138</td>
<td>4d2c667b</td>
<td>53380d13</td>
<td>650a7354</td>
<td>766a0bb7</td>
<td>91c2c92e</td>
<td>94a0b92b</td>
</tr>
<tr>
<td>a2bfe8a1</td>
<td>ab47e4eb</td>
<td>c24b8b70</td>
<td>c76c51a3</td>
<td>d192e819</td>
<td>d6990624</td>
<td>f40e35b5</td>
<td>0f0e47e3</td>
</tr>
<tr>
<td>19a4c116</td>
<td>6e376c08</td>
<td>2748774c</td>
<td>34b0c8c8</td>
<td>3910bcb3</td>
<td>4ed8aa4a</td>
<td>5b9ca4af</td>
<td>682e6f39</td>
</tr>
<tr>
<td>74f02f3b</td>
<td>75a5636f</td>
<td>04c87014</td>
<td>0c70208</td>
<td>0b2effa</td>
<td>4506ceb</td>
<td>bef9a3fb</td>
<td>c67178f3</td>
</tr>
</tbody>
</table>

Figure 4: SHA256 Message Decomposition and Padding
Table 6: SHA256 Initial Hash Values

<table>
<thead>
<tr>
<th>Hash Double-Word</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H^0_1$</td>
<td>6a09e667</td>
</tr>
<tr>
<td>$H^0_2$</td>
<td>bb67a85</td>
</tr>
<tr>
<td>$H^0_3$</td>
<td>3c6ef372</td>
</tr>
<tr>
<td>$H^0_4$</td>
<td>a54ff53a</td>
</tr>
<tr>
<td>$H^0_5$</td>
<td>510e527f</td>
</tr>
<tr>
<td>$H^0_6$</td>
<td>9b0568c</td>
</tr>
<tr>
<td>$H^0_7$</td>
<td>1f83d9ab</td>
</tr>
<tr>
<td>$H^0_8$</td>
<td>5be0cd19</td>
</tr>
</tbody>
</table>

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](image)

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 $\sigma_0$ and $\sigma_1$ referred to as the Sigma 0 and Sigma 1 functions.

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

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 $H^0_1$ through $H^0_8$ in the first hash’s first round [5].

$$a_{i-1}, b_{i-1}, c_{i-1}, d_{i-1}, e_{i-1}, f_{i-1}, g_{i-1}, h_{i-1} = H_{i-1}^0, H_{i-1}^1, H_{i-1}^2, H_{i-1}^3, H_{i-1}^4, H_{i-1}^5, H_{i-1}^6, H_{i-1}^7$$

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 $\Sigma_0$ and $\Sigma_1$ referred to as the Sum 0 and Sum 1 functions [5].

$$T_1 = h + \Sigma_1(e_{i-1}) + CH(e_{i-1}, f_{i-1}, g_{i-1}) + K_i + W_i$$

$$T_2 = \Sigma_0(a_{i-1}) + MAJ(a_{i-1}, b_{i-1}, c_{i-1})$$

$$h = g_{i-1}$$

$$g = f_{i-1}$$

$$f = e_{i-1}$$

$$e = d_{i-1} + T_1$$

$$d = c_{i-1}$$

$$c = b_{i-1}$$

$$b = a_{i-1}$$

$$a = T_1 + T_2$$

where:

$$CH(X, Y, Z) = (X \& Y) \oplus (~X \& Z)$$

$$MAJ(X, Y, Z) = (X \& Y) \oplus (X \& Z) \oplus (Y \& Z)$$

$$\Sigma_0(X) = ROR(X, 2) \oplus ROR(X, 13) \oplus ROR(X, 22)$$

$$\Sigma_1(X) = ROR(X, 6) \oplus ROR(X, 11) \oplus ROR(X, 25)$$
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].

\[
\begin{align*}
H'_1 &= H'_{i-1} + a \\
H'_2 &= H'_{i-1} + b \\
H'_3 &= H'_{i-1} + c \\
H'_4 &= H'_{i-1} + d \\
H'_5 &= H'_{i-1} + e \\
H'_6 &= H'_{i-1} + f \\
H'_7 &= H'_{i-1} + g \\
H'_8 &= H'_{i-1} + h
\end{align*}
\] (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].

\[
H = \{H'_1, H'_2, H'_3, H'_4, H'_5, H'_6, H'_7, H'_8\}
\] (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>
<thead>
<tr>
<th>Instruction Type</th>
<th>FUNC6</th>
<th>VM</th>
<th>VS2</th>
<th>Operand 1</th>
<th>FUNC3</th>
<th>VD</th>
<th>OPCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPIVV</td>
<td>000</td>
<td>VS1</td>
<td></td>
<td>01010111</td>
<td>10001</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>OPIVI</td>
<td>001</td>
<td>IMM</td>
<td></td>
<td>01101111</td>
<td>10001</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>OPIVX</td>
<td>010</td>
<td>RS1</td>
<td></td>
<td>10111111</td>
<td>10001</td>
<td>01</td>
<td></td>
</tr>
</tbody>
</table>

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.
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 {@} 0b0000; /* 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] ^ rot32(mixed, unsigned(shamt));  
    X(rd) = exts(result); RETIRE_SUCCESS 
} 

function clause execute (AES32ESI (bs, rs2, rs1, rd)) {  
    let shamt : bits(5) = bs {@} 0b0000; /* shamt = bs*8 */  
    let si : bits(8) = X(rs2)[31..0] >> shamt)[7..0]; /* SBox Input */  
    let so : bits(32) = 0x0000000 @ aes_sbox_fwd(si);  
    let result : bits(32) = X(rs1)[31..0] ^ rot32(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].

<table>
<thead>
<tr>
<th>Middle Round</th>
<th>Last Round</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes32esmi</td>
<td>aes32esi</td>
</tr>
<tr>
<td>a0, a0, t0, 0</td>
<td>a0, a0, t0, 0</td>
</tr>
<tr>
<td>aes32esmi</td>
<td>aes32esi</td>
</tr>
<tr>
<td>a0, a0, t1, 1</td>
<td>a0, a0, t1, 1</td>
</tr>
<tr>
<td>aes32esmi</td>
<td>aes32esi</td>
</tr>
<tr>
<td>a0, a0, t2, 2</td>
<td>a0, a0, t2, 2</td>
</tr>
<tr>
<td>aes32esmi</td>
<td>aes32esi</td>
</tr>
<tr>
<td>a0, a0, t3, 3</td>
<td>a0, a0, t3, 3</td>
</tr>
<tr>
<td>aes32esi</td>
<td>aes32esi</td>
</tr>
<tr>
<td>a1, a1, t1, 0</td>
<td>a1, a1, t1, 0</td>
</tr>
<tr>
<td>aes32esi</td>
<td>aes32esi</td>
</tr>
<tr>
<td>a1, a1, t2, 1</td>
<td>a1, a1, t2, 1</td>
</tr>
<tr>
<td>aes32esi</td>
<td>aes32esi</td>
</tr>
<tr>
<td>a1, a1, t3, 2</td>
<td>a1, a1, t3, 2</td>
</tr>
<tr>
<td>aes32esi</td>
<td>aes32esi</td>
</tr>
<tr>
<td>a1, a1, t0, 3</td>
<td>a1, a1, t0, 3</td>
</tr>
<tr>
<td>aes32esi</td>
<td>aes32esi</td>
</tr>
<tr>
<td>a2, a2, t0, 2</td>
<td>a2, a2, t0, 2</td>
</tr>
<tr>
<td>aes32esi</td>
<td>aes32esi</td>
</tr>
<tr>
<td>a2, a2, t3, 1</td>
<td>a2, a2, t3, 1</td>
</tr>
<tr>
<td>aes32esi</td>
<td>aes32esi</td>
</tr>
<tr>
<td>a2, a2, t0, 2</td>
<td>a2, a2, t0, 2</td>
</tr>
<tr>
<td>aes32esi</td>
<td>aes32esi</td>
</tr>
<tr>
<td>a2, a2, t1, 3</td>
<td>a2, a2, t1, 3</td>
</tr>
<tr>
<td>aes32esi</td>
<td>aes32esi</td>
</tr>
<tr>
<td>a3, a3, t3, 0</td>
<td>a3, a3, t3, 0</td>
</tr>
<tr>
<td>aes32esi</td>
<td>aes32esi</td>
</tr>
<tr>
<td>a3, a3, t0, 1</td>
<td>a3, a3, t0, 1</td>
</tr>
<tr>
<td>aes32esi</td>
<td>aes32esi</td>
</tr>
<tr>
<td>a3, a3, t1, 2</td>
<td>a3, a3, t1, 2</td>
</tr>
<tr>
<td>aes32esi</td>
<td>aes32esi</td>
</tr>
<tr>
<td>a3, a3, t2, 3</td>
<td>a3, a3, t2, 3</td>
</tr>
</tbody>
</table>

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].
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].

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>
<thead>
<tr>
<th>Instruction</th>
<th>BS</th>
<th>FUNC5</th>
<th>RS2</th>
<th>RS1</th>
<th>FUNC3</th>
<th>RD</th>
<th>OPCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>sha256sig0</td>
<td>0</td>
<td>0 0 1 0 0 0 0 0 1 0</td>
<td></td>
<td>0 0 1</td>
<td>0 0 1 0 0 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sha256sig1</td>
<td>0</td>
<td>0 0 1 0 0 0 0 0 1 1</td>
<td></td>
<td>0 0 1</td>
<td>0 0 1 0 0 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sha256sum0</td>
<td>0</td>
<td>0 0 1 0 0 0 0 0 0 0</td>
<td></td>
<td>0 0 1</td>
<td>0 0 1 0 0 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sha256sum1</td>
<td>0</td>
<td>0 0 1 0 0 0 0 0 0 1</td>
<td></td>
<td>0 0 1</td>
<td>0 0 1 0 0 1 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

\[
\text{sha256sig0 rd, rsi} \\
\text{sha256sig1 rd, rsi}
\]

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

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

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.

\[
\begin{align*}
\text{add} & \quad t0, a0, a2 \quad // \text{add } W[i-16], W[i-7] \\
\text{sha256sig0} & \quad t1, a1 \quad // \text{sig0 } W[i-15] \\
\text{add} & \quad t0, t0, t1 \quad // \text{add to total} \\
\text{sha256sig1} & \quad t1, a3 \quad // \text{sig1 } W[i-2] \\
\text{add} & \quad t0, t0, t1 \quad // \text{new double-word} \\
\text{sw} & \quad t0, 64(sp) \quad // \text{store new double-word} \\
\text{lw} & \quad a0, 4(sp) \quad // \text{load } W[i-16] \\
\text{lw} & \quad a1, 8(sp) \quad // \text{load } W[i-15] \\
\text{lw} & \quad a2, 40(sp) \quad // \text{load } W[i-7] \\
\text{lw} & \quad a3, 60(sp) \quad // \text{load } W[i-2]
\end{align*}
\]

Figure 14: RISC-V SHA256 sig0 and sig1 Instructions Assembly Use Case
The reason the scalar instructions target 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 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.

\[
\text{sha256sum0 } \text{rd}, \text{rs1} \\
\text{sha256sum1 } \text{rd}, \text{rs1}
\]

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

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

```plaintext
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 $H^i_1$ through $H^i_8$. 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 s4, 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 (AESENCE, 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].

![Pseudocode for Intel's AES Instructions](image18.png)

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.
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 $H_i^{t-1}$ through $H_0^{t-1}$ 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**

<table>
<thead>
<tr>
<th>Register</th>
<th>Double-word</th>
</tr>
</thead>
<tbody>
<tr>
<td>X3</td>
<td>( w[i-1] )</td>
</tr>
<tr>
<td></td>
<td>( w[i-2] )</td>
</tr>
<tr>
<td></td>
<td>( w[i-3] )</td>
</tr>
<tr>
<td></td>
<td>( w[i-4] )</td>
</tr>
<tr>
<td>X2</td>
<td>( w[i-5] )</td>
</tr>
<tr>
<td></td>
<td>( w[i-6] )</td>
</tr>
<tr>
<td></td>
<td>( w[i-7] )</td>
</tr>
<tr>
<td></td>
<td>( w[i-8] )</td>
</tr>
<tr>
<td>X1</td>
<td>( w[i-9] )</td>
</tr>
<tr>
<td></td>
<td>( w[i-10] )</td>
</tr>
<tr>
<td></td>
<td>( w[i-11] )</td>
</tr>
<tr>
<td></td>
<td>( w[i-12] )</td>
</tr>
<tr>
<td>X0</td>
<td>( w[i-13] )</td>
</tr>
<tr>
<td></td>
<td>( w[i-14] )</td>
</tr>
<tr>
<td></td>
<td>( w[i-15] )</td>
</tr>
<tr>
<td></td>
<td>( w[i-16] )</td>
</tr>
</tbody>
</table>

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].
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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>VC</th>
<th>VM</th>
<th>VS2</th>
<th>VS1</th>
<th>VC</th>
<th>VD</th>
<th>OPCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>vaesds.w</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1 0 1 0 1 1 1</td>
</tr>
<tr>
<td>vaesdsn.w</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1 0 1 0 1 1 1</td>
</tr>
<tr>
<td>vaeses.w</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1 0 1 0 1 1 1</td>
</tr>
<tr>
<td>vaesem.w</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1 0 1 0 1 1 1</td>
</tr>
</tbody>
</table>

3.2.1 VAESES and VAES

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.

```assembly
vaesem.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.
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
vaesem.vv v0, v0, v2  //round 1
vaesem.vv v0, v0, v3  //round 2
vaesem.vv v0, v0, v4  //round 3
vaesem.vv v0, v0, v5  //round 4
vaesem.vv v0, v0, v6  //round 5
vaesem.vv v0, v0, v7  //round 6
vaesem.vv v0, v0, v8  //round 7
vaesem.vv v0, v0, v9  //round 8
vaesem.vv v0, v0, v10 //round 9
vaesem.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.

```
vaesdsam.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.

```plaintext
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.
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>
<thead>
<tr>
<th>Instruction</th>
<th>FUNC6</th>
<th>VM</th>
<th>VS2</th>
<th>VS1</th>
<th>DWS</th>
<th>VD</th>
<th>OPCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>vsha256sch.w</td>
<td>1 1 0 1 1 0</td>
<td></td>
<td></td>
<td></td>
<td>0 0 0</td>
<td>1 0 1 0 1 1 1</td>
<td></td>
</tr>
<tr>
<td>vsha256hash.w</td>
<td>1 1 0 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 0 1 0 1 1 1</td>
<td></td>
</tr>
</tbody>
</table>

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.

\[ \text{vsha256sch.vv vd, vs2, vs1, vm} \]

\textit{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(x3[127..96]);
    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.

```
vhsha256sch.vv v2, v0, v1  //generate double-words 17-24
vhsha256sch.vv v3, v1, v2  //generate double-words 25-32
vhsha256sch.vv v4, v2, v3  //generate double-words 33-40
vhsha256sch.vv v5, v3, v4  //generate double-words 41-48
vhsha256sch.vv v6, v4, v5  //generate double-words 49-56
vhsha256sch.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.
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.

```plaintext
function clause execute (VSHA256HA38( 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] + sig(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.

\[
\begin{align*}
vsha256hash.vv & \ v0, \ v2, \ v1, \ 0 \ //\text{Hash round } i \\
vsha256hash.vv & \ v0, \ v2, \ v1, \ 1 \ //\text{Hash round } i+1 \\
vsha256hash.vv & \ v0, \ v2, \ v1, \ 2 \ //\text{Hash round } i+2 \\
vsha256hash.vv & \ v0, \ v2, \ v1, \ 3 \ //\text{Hash round } i+3 \\
vsha256hash.vv & \ v0, \ v2, \ v1, \ 4 \ //\text{Hash round } i+4 \\
vsha256hash.vv & \ v0, \ v2, \ v1, \ 5 \ //\text{Hash round } i+5 \\
vsha256hash.vv & \ v0, \ v2, \ v1, \ 6 \ //\text{Hash round } i+6 \\
vsha256hash.vv & \ v0, \ v2, \ v1, \ 7 \ //\text{Hash round } i+7
\end{align*}
\]

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 vaesdsms 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_VAES 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.
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 vaese 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 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>1</td>
</tr>
<tr>
<td>res[3:0]</td>
<td>ALU_VAES, ALU_VAES_ES</td>
</tr>
<tr>
<td>aese[2:0]</td>
<td></td>
</tr>
<tr>
<td>operand1_32[127:0]</td>
<td>000000000000</td>
</tr>
<tr>
<td>operand2_32[127:0]</td>
<td>000000000000</td>
</tr>
<tr>
<td>shiftrow[127:0]</td>
<td>000000000000</td>
</tr>
<tr>
<td>sbo[127:0]</td>
<td>000000000000</td>
</tr>
<tr>
<td>mixo[127:0]</td>
<td>000000000000</td>
</tr>
<tr>
<td>roundo[127:0]</td>
<td>000000000000</td>
</tr>
<tr>
<td>crypto_tmp_d[127:0]</td>
<td>000000000000</td>
</tr>
</tbody>
</table>

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.
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.

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

<table>
<thead>
<tr>
<th>Vector Assembly</th>
<th>Scalar Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>vaesesm.vv v0, v0, v1 //round 1</td>
<td>aes32esmi a0, a0, t0, 0</td>
</tr>
<tr>
<td></td>
<td>aes32esmi a0, a0, t1, 1</td>
</tr>
<tr>
<td></td>
<td>aes32esmi a0, a0, t2, 2</td>
</tr>
<tr>
<td></td>
<td>aes32esmi a0, a0, t3, 3</td>
</tr>
<tr>
<td></td>
<td>aes32esmi a1, a1, t1, 0</td>
</tr>
<tr>
<td></td>
<td>aes32esmi a1, a1, t2, 1</td>
</tr>
<tr>
<td></td>
<td>aes32esmi a1, a1, t3, 2</td>
</tr>
<tr>
<td></td>
<td>aes32esmi a1, a1, t0, 3</td>
</tr>
<tr>
<td></td>
<td>aes32esmi a2, a2, t2, 0</td>
</tr>
<tr>
<td></td>
<td>aes32esmi a2, a2, t3, 1</td>
</tr>
<tr>
<td></td>
<td>aes32esmi a2, a2, t0, 2</td>
</tr>
<tr>
<td></td>
<td>aes32esmi a2, a2, t1, 3</td>
</tr>
<tr>
<td></td>
<td>aes32esmi a3, a3, t3, 0</td>
</tr>
<tr>
<td></td>
<td>aes32esmi a3, a3, t0, 1</td>
</tr>
<tr>
<td></td>
<td>aes32esmi a3, a3, t1, 2</td>
</tr>
<tr>
<td></td>
<td>aes32esmi a3, a3, t2, 3 //round 1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>706F7274</th>
<th>736D6F75</th>
<th>746B8000</th>
<th>00000000</th>
<th>00000000</th>
<th>00000000</th>
<th>00000000</th>
<th>00000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000050</td>
</tr>
<tr>
<td>2FC59E5F</td>
<td>A1F54B8F</td>
<td>F15B037D</td>
<td>0C9E5996</td>
<td>01ED1087</td>
<td>E7FAE04A</td>
<td>2A531BDF</td>
<td>5BDB7019</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8091D44B</td>
<td>60B9CC0F</td>
<td>FABEF733</td>
<td>ECD282D4</td>
<td>3045E214</td>
<td>F62E8215</td>
<td>F27D9B62</td>
<td>54A70545</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>1</td>
</tr>
<tr>
<td>res[3:0]</td>
<td>ALU_VSHA</td>
</tr>
<tr>
<td>aes[2:0]</td>
<td>ALU_VSHA_CHK</td>
</tr>
<tr>
<td>operand1_32[255:0]</td>
<td>9d6deb133caf2</td>
</tr>
<tr>
<td>operand2_32[255:0]</td>
<td>776ad92d23502</td>
</tr>
<tr>
<td>dw[1][31:0]</td>
<td>8091d44b</td>
</tr>
<tr>
<td>dw[2][31:0]</td>
<td>69b9cc2f</td>
</tr>
<tr>
<td>dw[3][31:0]</td>
<td>fabef735</td>
</tr>
<tr>
<td>dw[4][31:0]</td>
<td>ecd28e6d</td>
</tr>
<tr>
<td>dw[5][31:0]</td>
<td>3045e214</td>
</tr>
<tr>
<td>dw[6][31:0]</td>
<td>f62e8215</td>
</tr>
<tr>
<td>dw[7][31:0]</td>
<td>f27d9b62</td>
</tr>
<tr>
<td>dw[8][31:0]</td>
<td>54a70545</td>
</tr>
<tr>
<td>crypto_tmp_d[255:0]</td>
<td>54a70545f27d6</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Vector Assembly</th>
<th>Scalar Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>vsha256sch.vv v2, v0, v1 //8 new DWs</td>
<td>add t0, a0, a2</td>
</tr>
<tr>
<td>sha256sch0 t1, a1</td>
<td>add t0, t0, t1</td>
</tr>
<tr>
<td>sha256sch1 t1, a3</td>
<td>add a4, t0, t1 //1 new DW</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>x8</td>
</tr>
</tbody>
</table>
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


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_VAES,
   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;  //first generated SHA256 double-word
logic [127:0] x1;  //second generated SHA256 double-word
logic [127:0] x2;  //third generated SHA256 double-word
logic [127:0] x3;  //fourth generated SHA256 double-word
logic [31:0] dw1;  //first generated SHA256 double-word
logic [31:0] dw2;  //second 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
    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;
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(x3[127:96]));
    dw4=x0[127:96]+x3[31:0]+(`Sig0(x1[127:96]))+(`Sig1(x2[31:0]));
    dw5=x1[31:0]+x3[63:32]+(`Sig0(x1[63:32]))+(`Sig1(x2[31:0]));
    dw6=x1[63:32]+x3[95:64]+(`Sig0(x1[95:64]))+(`Sig1(x2[31:0]));
    dw7=x1[95:64]+x3[127:96]+(`Sig0(x1[127:96]))+(`Sig1(x3[127:96]));
    dw8=x1[127:96]+x3[127:96]+(`Sig0(x2[31:0]))+(`Sig1(x3[127:96]));
    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;  // ALU operand width in bits
        s1=sbo[(32 * i)-16 :) 8;  // ALU operand width in bits
        s2=sbo[(32 * i)-24 :) 8;  // ALU operand width in bits
        s3=sbo[(32 * i)-32 :) 8;  // ALU operand width in bits
        mixo[(32 * i)-8 :] = ((s0<<1)^s0[7]) ? 8'h1b : 8'h0)^s1^s1[7] ? 8'h1b : 8'h0)^s2^s2[7] ? 8'h1b : 8'h0)^s3^s3[7] ? 8'h1b : 8'h0);  // ALU operand width in bits
        mixo[(32 * i)-16 :] = s0^s1^s2<<1)^s2[7] ? 8'h1b : 8'h0)^s3^s3[7] ? 8'h1b : 8'h0);  // ALU operand width in bits
        mixo[(32 * i)-24 :] = s0^s1^s2<<1)^s2[7] ? 8'h1b : 8'h0)^s3^s3[7] ? 8'h1b : 8'h0);  // ALU operand width in bits
        mixo[(32 * i)-32 :] = s0^s1^s2<<1)^s2[7] ? 8'h1b : 8'h0)^s3^s3[7] ? 8'h1b : 8'h0);  // ALU operand width in bits
    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,  
    output logic [ALU_OP_W-1:0] so,  
    output logic [ALU_OP_W-1:0] sm4,  
    output logic [ALU_OP_W-1:0] dec,  
    output logic [ALU_OP_W-1:0] 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
.globl main
main:
    la     sp, vdata_start
    li     t0, 4
    vsetvli t0, t0, e32,ml,lu,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
.globl vdata_start
vdata_start:
    .word  0x2054776F //Data
    .word  0xE096E65
    .word  0x4F6E6520
    .word  0x54776E20
    .word  0x67204675 //Round key 0
    .word  0x204B756F
    .word  0x3206D79
    .word  0x4686174
    .word  0x679A293 //Round key 1
    .word  0xB159E4E6
    .word  0x91129188
    .word  0xE232FCF1

64
.word 0x00000000  //Round key 2
.word 0x00000000
.word 0x00000000
.word 0x00000050

.word 0xC3031EFB  //Round key 3
.word 0x6339E901
.word 0xB157ABC6
.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 0xB66C9527
.word 0xBB87C4715
.word 0xBD3DC2B7

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

.word 0x56954B6C  //Round key 8
.word 0xE43D7A06
.word 0x8FB84522
.word 0x8E51EF21

.word 0xF7F1CBD8  //Round key 9
.word 0xA16480B4
.word 0x4559FAB2
.word 0xBF62BF90

.word 0x3B316F26  //Round key 10
.word 0xCCC0A4FE
.word 0x6DA4244A
.word 0x29FDDEF8
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 0x00000000
  .word 0x000000050