Leveraging RISC-V Vector Instructions for AES-128 Encryption

Authors:

Azka Aqeel - 27068

Institute of Business Administration

Karachi, Pakistan

Abstract- This paper presents the implementation of the AES-128 encryption algorithm using RISC-V vector instructions on the Veer simulator, running on a Linux machine. By leveraging the parallel processing capabilities of RISC-V's vector extensions, we detail the methods used to optimize the AES-128 encryption process. The implementation approach is thoroughly explained, highlighting how vector instructions are utilized to enhance efficiency and performance.

I. INTRODUCTION

A. *Advanced Encryption Standard*

The Advanced Encryption Standard (AES) is a cornerstone of modern cryptography, providing robust security and efficiency in data encryption. Developed by the U.S. National Institute of Standards and Technology (NIST), AES replaced the aging Data Encryption Standard (DES) in 2001, following a rigorous selection process that ensured its cryptographic strength and practical applicability. AES operates as a symmetric-key encryption algorithm, employing fixed-size blocks and varying key lengths to secure sensitive data across diverse computing environments.

B. *RISC-V Vector Architecture*

RISC-V, an open-source instruction set architecture (ISA), has attracted considerable attention in recent years due to its simplicity, flexibility, and scalability. The acronym "RISC-V" stands for "Reduced Instruction Set Computing - Version 5." Unlike proprietary ISAs, RISC-V offers a customizable framework accessible to all, empowering researchers and developers to innovate and experiment with novel computing designs. Its modular design and support for custom extensions create fertile ground for exploring new approaches to hardware design and optimization. The RISC-V Vector Architecture is an extension to the RISC-V instruction set architecture (ISA) designed to support vector processing capabilities. In traditional scalar architectures, instructions operate on individual data elements, while vector architectures allow a single instruction to perform operations on multiple data elements simultaneously, known as vectorization. This approach is particularly beneficial for data-intensive tasks such as signal processing, image processing, and cryptography, where large amounts of data need to be processed in parallel.

C. *Setup of Tools on Linux*

VeeR-ISS (Vector Extension for RISC-V - Instruction Set Simulator) is a simulation environment designed specifically for experimenting with RISC-V vector instructions. It enables developers to write, compile, and simulate RISC-V vectorized code, facilitating the evaluation of vector instruction performance and behavior. For this project, we set up VeeR-ISS on our Linux system, simplifying the process by the manual compilation from the official repository source code.

II. RELATED WORK

1. *AES Implementations on Vector Architectures*

Research published in IEEE Transactions on Computers by O'Connor and Baugh [1], presents efficient implementations of AES encryption on vector architectures. The study demonstrates significant performance improvements compared to scalar implementations, showcasing the potential of vectorized approaches for accelerating AES encryption.

2. *Vectorized AES Encryption for High-Performance Platforms:*

Research presented at the ACM International Conference on Supercomputing by Wang, Zhang, and Li [2] introduces a vectorized AES encryption scheme optimized for high-performance and low-energy platforms. The study showcases the effectiveness of vector processing in improving both speed and energy efficiency for cryptographic computations.

III. METHODOLOGY

1. *Overview of AES*

Advanced Encryption Standard (AES) employs a methodology where data is encrypted in fixed-size blocks of 128 bits. The AES allows for key sizes of 128, 192, or 256 bits, affecting the number of encryption rounds—10, 12, or 14, respectively. Each plaintext input to the algorithm is divided into blocks, with each block consisting of 128 bits

(16 bytes). These blocks are independently processed during encryption, with the last block potentially padded to ensure its size is a multiple of 128 bits. The encryption and decryption processes operate on data in the form of a two-dimensional array known as the *state array*. For AES-128, the state array comprises four rows and four columns, totaling 16 bytes (128 bits). Each byte within the state array represents an element of the plaintext or ciphertext data being processed, facilitating the encryption and decryption operations

A. Encryption Process Steps

Each encryption round comprises of the following steps except for the last round in which we skip the Mix-columns step:

Key Expansion: The initial key is used for Round 0. For rounds 1-10, we generate keys by expanding the first key. The 128-bit key is divided into 4 words, columns, on which we apply a series of operations. The last column is rotated upwards by 1 byte. Substitution follows, replacing each byte of data with another byte according to a fixed substitution table known as the S-box (Substitution-box). This column is then XORed with the column 4 locations earlier to it, and a round constant value. This generates the first column/word of our next key. The next 3 columns are generated in a manner where the ith column/word is produced by XORing the (i-1)th column with the ith column/word in the last generated key. For this iterative process, we have 11 round constants.

Note: It is a programmer's choice to either generate all the keys before the 10 rounds or generate keys in each encryption round. We have adopted the choice to generate one keys per round for encryption, and for decryption we generated all the keys prior to the decryption rounds.

AddRoundKey: For Round 0, we add the initial key to the plaintext to generate the state array. For subsequent rounds, we add the next key in expansion to the state array. The addition takes place by XORing each element of both matrices.

SubBytes: For this step, we have a fixed S-box containing 256 bytes hard-coded in memory. Each element in the state array is substituted with another byte based on a fixed substitution table called the S-box (Substitution-box). The value of each element becomes the memory address of the new value we access in the S-box. To ensure that it is within bounds, we add it to the base address of the S-box.

ShiftRows: The bytes within each column are shifted cyclically to the left, with the number of shifts varying based on the row index. In the first row, no shifts are applied. In the second row, each byte is shifted to the left by one position. In the third row, each byte is shifted by two positions, and in the fourth row, each byte is shifted by three positions.

MixColumns: The MixColumns transformation operates on the columns of the state array matrix, treating each column as a polynomial over the finite field. Specifically, each byte in the column is multiplied by a fixed polynomial matrix, known as the MixColumns matrix, using Galois multiplication. The multiplication is done using polynomial multiplication in GF(2^8), followed by reduction modulo the irreducible polynomial `x^8 + x^4 + x^3 + x + 1`

B. Decryption Process Steps

AES decryption involves reversing the encryption process by applying inverse transformations. This includes key expansion, SubBytes, ShiftRows, and MixColumns, performed in reverse order to recover the original plaintext from the ciphertext.

Key Expansion: This process remains the same as encryption Key Expansion.

InverseAddRoundKey: For Round 0, we add the last key produced in the key expansion. For subsequent rounds, we use the second to last key in expansion to the state array. The addition takes place by XORing each element of both matrices.

InverseShiftRows: The bytes within each column are shifted cyclically to the right, with the number of shifts varying based on the row index. In the first row, no shifts are applied. In the second row, each byte is shifted to the right by one position. In the third row, each byte is shifted by two positions, and in the fourth row, each byte is shifted by three positions.

InverseSubBytes: This step involves cyclically shifting the bytes within each column to the right, with the number of shifts varying based on the row index. Specifically, no shifts are applied in the first row, while in the second row, each byte is shifted to the right by one position. In the third row, each byte is shifted by two positions to the right, and in the fourth row, each byte is shifted by three positions to the right.

InverseMixColumns: Each column is treated as a polynomial over the finite field, and each byte within the column is multiplied by a fixed polynomial matrix (different than what was used for encryption) using Galois multiplication. This multiplication, involving polynomial multiplication modulo an irreducible polynomial in the finite field GF(2^8), contributes to reversing the encryption process and recovering the original plaintext from the ciphertext.

2. *Vectorization Strategy*

In examining the AES encryption and decryption processes above, it becomes evident that during the "AddRoundKey" and "SubBytes" stage, all 16 elements of the key and state array undergo the same treatment. Similarly, in subsequent rounds, a minimum of four elements are treated uniformly, thereby diminishing the necessity of instructions from 16 to 4 or even to 1, to address a collective total of 16 elements.

We can leverage the capabilities of the RISC-V vector architecture and its corresponding ISA to achieve this task with fewer instructions. Multiple data elements can be loaded from memory to Vector Registers simultaneously. Vector registers are specialized hardware registers used in vector processing architectures, such as RISC-V Vector ISA. These registers can hold multiple data elements simultaneously, organized as vectors. Unlike scalar registers, which typically hold single data elements like integers or floating-point numbers, vector registers can store several data elements in a contiguous fashion.

In vectorized processing, data is processed in parallel using vector instructions. These instructions operate on entire vectors stored in vector registers, performing the same operation on multiple data elements concurrently. For example, instead of executing addition operations individually on each element of an array, a vectorized instruction can add corresponding elements of two vectors in a single operation.

The data processed using vectorization is stored in memory and loaded into vector registers before performing vectorized operations. Once the data is loaded into the vector registers, vector instructions operate on these registers, enabling efficient parallel processing of data elements. This approach significantly accelerates computation-intensive tasks by exploiting parallelism at the hardware level.

IV. Implementation

We successfully leveraged the Vector-ISA to vectorize all steps involved in both AES encryption and decryption.

A. Encryption

Vectorized Key Expansion: In this iterative process, a column containing 4 elements of 1 byte each can be treated as a vector as all 4 elements undergo the same operations. We make use of the instruction *vsetvli x1, x2, e8* to set a vector register which processes 4 elements of 8 byte each. We load the 4 elements from memory using the vector load instruction vle8.v v1, (s1) where v1 is the vector register we load into, s1 provides the base address of the key expansion array in memory, from where we load 4 elements stored contiguously in memory.

The next step is rotation in the last column, each byte is rotated upwards by 1 byte. Since this is a wraparound rotation, the byte at the top is placed at the bottom most location. For this rotation, we have predefined a rotation offset 4x1 matrix in memory that stores the following offsets: *keyRotationOffset: .byte 1, 2, 3, 0.* These offsets are first loaded to a vector register. The ith element/byte in it is added to the base address of the of key and then used as memory addresses for loading the ith element of the rotated column, where i is 0, 1, 2, 3, each referring to an element in the column. This is successfully done by using Vector indexed-unordered load instruction *vluxei8.v vd, (rs1), vs2* where vd serves as the destination register in which values are loaded, rs1 holds the base address of the key expansion array, and vs2 holds the rotation offsets. This successfully loads elements in wraparound rotated fashion.

The next steps include XORing of a column with another and for this we first load the words from memory in separate registers using vector load *vle8.v* instruction. Then we perform the XOR operation by using the vectorized *vxor.vv vd, v1, v2* which takes 2 source vector registers and performs XOR between their respective elements.

The resulting columns are stored in memory using the vector store instruction. The processed word consisting of 4 bytes is stored using *vse8.v vs, (rs1)* where vs is the source vector register which we wish to load at the base address rs1 in memory. In our case, rs1 would be the base address of that column in key.

Vectorized Sub-Bytes:

All 16 elements of the state array are processed similarly; therefore, we load them in a single vector register. Upon addition to the base address of the S-Box, each element yields the memory address of a new element within the S-Box. This new element is subsequently loaded and utilized as a replacement. To perform this task, we make use of the Vector indexed-unordered load instruction *vluxei8.v vd, (rs1), vs2* where vd serves as the destination register in which the new elements are loaded from S-Box, rs1 holds the base address of the S-Box, and vs2 is the vector register containing the 16 bytes from the state array. This instruction functions such that each new element's memory location is found by adding the last element to base address of S-box, and then loading it to the destination register.

Vectorized Shift Rows:

Our data storage in memory is such that 4 contiguous bytes in memory are elements of 1 column. We cannot use shift functions directly as elements of a single row are 4 locations apart in memory and vectorized load or store works on contiguous data in memory. This restricted our approach to using an alternate method. We calculated offsets necessary for shifting. To visualize this, look at the figure below.

$$Before\,ShiftRows: \begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{bmatrix}$$

Figure 1.0, Matrix representation of the State Array elements

It visualizes the contiguous data in memory as the 4x4 state array we are dealing with. The values indicate the offset from the base address of this array. That is, the element at the [1, 0] entry stores 1 because that is its offset from the base address from the memory. So, to access this element, I need to add 1 to the base address of the array to achieve its memory address. In the first row, no shifts are applied. In the second row, each byte is shifted to the left by one position. In the third row, each byte is shifted by two positions, and in the fourth row, each byte is shifted by three positions. The matrix representation will be as follows:

$$After\,ShiftRows: \begin{bmatrix} 0 & 4 & 8 & 12 \\ 5 & 9 & 13 & 1 \\ 10 & 14 & 2 & 6 \\ 15 & 3 & 7 & 11 \end{bmatrix}$$

Figure 1.1, Matrix Representation of the State Array elements

The matrix representation after shifting serves as the Shift Offset Matrix. To find the resulting matrix of ShiftRows, we set a vector register that would contain the resulting matrix of 16 elements. To load each element, we add its respective offset to the base address of the state array. This results in the memory location of the element which will come in the current location of the result matrix. We smoothly execute this process by first loading all 16 elements of the ShiftOffset in a single vector register. Then we make use of the Vector indexed-unordered load instruction *vluxei8.v vd, (rs1), vs2* where vd serves as the destination register which will hold the result of ShiftRows, rs1 holds the base address of the State Array, and vs2 is the vector register containing the 16 bytes of ShiftOffest.

The resulting vector is then stored in memory using vectorized store instruction that simultaneously stores all 16 elements in memory.

Vectorized Mix-Columns:

The MixColumns step involves multiplying each column of the state array by the fixed MixColumns matrix. The multiplication is done using polynomial multiplication in GF(2^8), followed by reduction modulo the irreducible polynomial `x^8 + x^4 + x^3 + x + 1`.

$$MixColumns\,matrix \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

Figure 1.3. Rijndael MixColumns's Matrix

This matrix is multiplied to the left of every column of the State Array, resulting in a new column. Each element of the column is multiplied by a corresponding element of the MixColumns matrix using Galois field multiplication.

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} s1 \\ s2 \\ s3 \\ s4 \end{bmatrix} = \begin{bmatrix} r1 \\ r2 \\ r3 \\ r4 \end{bmatrix}$$

Figure 1.4, Each byte in State Array is modulo multiplied in Rijndael's Galois Field by the MixColumns matrix.

Each byte of the resulting column is computed differently. The equations being:

$$r1 = 2.s1 \oplus 3.s2 \oplus 1.s3 \oplus 1.s4$$
$$r2 = 1.s1 \oplus 2.s2 \oplus 3.s3 \oplus 1.s4$$
$$r3 = 1.s1 \oplus 1.s2 \oplus 2.s3 \oplus 3.s4$$
$$r4 = 3.s1 \oplus 1.s2 \oplus 1.s3 \oplus 2.s4$$

Figure 1.5, Equations to compute each element of the new column

Multiplication by 1 means the byte remains unchanged. For multiplication by 2, we shift the byte left by 1 bit and check the 9th bit. If it is set, it means the result has exceeded the degree of the field which is 8. To ensure that the results stay within GF(2^8), we XOR the resulting byte with 0x1B if the 9th bit is set. For multiplication by 3, we XOR the result of multiplication by 2 with the original byte.

In this step, we see that each element in the column is treated differently. We cannot vectorize the multiplication for all elements of a column. However, we observe that we will need the result of each element's Galois multiplication by 2 and 3 in one of the four equations. Therefore, our first step was to perform Galois multiplication on all elements and store them in a vector. To do so, the column was loaded in a vector register using *vle8.v* instruction. Since we are dealing with elements of 8 bits each, there will be no way of checking whether the 9th bit will be set or not after shifting as the 9th bit will be discarded. Therefore, we check the most significant bit before shifting. This is done performing bitwise AND operation between each element and 0x80. We splat the scalar 0x80 to all active elements of a vector register by using the instruction *vmv.v.x vd, rs1* from the Vector Integer Move Instructions, where rs1 will be a register holding 0x80 and vd will be the vector register containing 0x80 in all its active elements. We then use *vand.vv vd, vs2, vs1* to perform bitwise AND operation between 0x80 and all elements. The resulting vector vd, in this case, will hold 0 if the 8th bit was not set. So, to filter that, we used Vector Integer Compare Instruction *vmsne.vi vd, vs2, imm* which writes 1 to the destination mask register element if the comparison between the resulting vector from the vand.vv and 0 evaluates to true, and 0 otherwise. The immediate value we used was 0. The resulting vector from this instruction is then used as vector mask to decide which elements need to be reduced by 0x1B. The vector mask allows for a smooth conditional XOR as it indicates which elements of the vector should be processed by a vector instruction. Now that our vector mask is ready, we shift all elements in the column by 1 bit using *vsll.vi vd, vs2, uimm.* We splat the scalar 0x1B to all active elements of a vector register by using the instruction *vmv.v.x vd, rs1* where rs1 will be a register holding 0x1B and vd will be the vector register containing 0x1B in all its active elements. We perform conditional XOR by using *vxor.vv vd, vs2, vs1* where the two source registers will be the resulting column after shifting and the vector holding 0x1B in all its active elements. This way we have successfully computed the Galois Multiplication by 2 results.

For Galois Multiplication by 3, we simply XOR the result of Galois Multiplication by 2 with the vector register holding the original column. The resulting vector holds the result.

Now that all the results needed for calculation of new column elements are ready, we simply need to copy all the results we need for a single equation to a single vector register. To do so, we have predefined four vector masks in our data section, each considering a single element active. This allows us to copy a single element from a vector. To copy an element, we have used Vector OR *vor.vv vd, vs2, vs1, vm* instruction where one of the source registers contains the element we need to copy, and the second source vector register contains all 0s. As per our need, we load the vector mask from memory in a vector register and use it as a vector mask during OR operation to copy only our desired element.

In this manner, we copy the four results needed for each equation. The next step is to perform XOR between all four elements in each vector to get r1, r2, r3, r4 shown in Figure 1.5. To perform XOR between all 4 elements in vector, we use the *vredxor.vs vd, vs2, vs1* instruction from the Vector Single-Width Integer Reduction Instructions. This instruction XORs all elements in the source register vs2 and vs1 and stores the result in the first element of destination register vd. For the second source register, we use a vector containing all 0s so that it does not affect our result. To create such a vector, we use vmv.v.x from Vector Integer Move Instructions instruction to splat 0 to all active elements. Since vredxor.vs stores result in the first element ny default, we need to move resulting values down to their place. To do so, we use the *vslideup.vi vd, vs2, uimm* from the Vector Slide Instructions. By specifying uimm, that is the number of positions each element should move upwards, we store the resulting elements in the right place.

We repeat this process for all four columns in a loop.

B. Decryption

For decryption we use the inverse of each process used in encryption, in the reverse. However, the vectorization strategy for both encryption and decryption is the same. The differences lie in the keys used as we use the expanded keys in reverse order for decryption rounds. The implementation of Sub-bytes also remains the same except that we use Inverse S-Box. In ShiftRows, instead of shifting to the left, rows are shifted to the right. Therefore, we hard-code a different shiftOffset matrix in memory, as shown below.

$$ShiftOffset\,Matrix \begin{bmatrix} 0 & 4 & 8 & 12 \\ 13 & 1 & 5 & 9 \\ 10 & 14 & 2 & 6 \\ 7 & 11 & 15 & 3 \end{bmatrix}$$

Figure 1.6, Shift Offset Matrix for decryption

For Inverse Mix Columns, the inverse matrix is used:

$$Inverse\,MixColumns \begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix}$$

Figure 1.7, Inverse MixColumns matrix for decryption, decimal values

While the vector strategy remains same, we need to perform Galois Multiplication by 9, 11, 13, 14. These can be broken down in terms of Galois Multiplication by 2. If we perform Galois Multiplication by 2 on the result of Galois Multiplication by 2 on a byte, we get the result of Galois Multiplication by 4 for that byte. If we are to repeat this again, we get the result of Galois Multiplication by 8.

For decryption, we computed the results for Galois Multiplication by 2, 4, and 8 for each column. To find the results for Galois Multiplication by 9, we simply needed to XOR the result of Galois multiplication by 8 with the original byte. For Galois multiplication by 11, we XORed the results of Galois multiplication by 8, 2, and the original byte. For Galois Multiplication by 13, we XORed the results of Galois Multiplication by 8, 4, and the original byte. Lastly, for Galois Multiplication by 14, we XORed the results of Galois Multiplication by 2, 4, and 8. These calculations were done for a column, i.e. 4 elements simultaneously using the vectorized XOR operations.

C. Vector ISA used

The table below lists the different Vector Instructions used. Their use has been explained in the implementation details above.

Instruction	Description	Implementation
vsetvli	Sets vector length and element width	To process all elements of state array or a column simultaneously
vle8.v	Loads 8-bit elements from memory	For loading data from memory e.g. plaintext
vse8.v	Stores 8-bit elements from a vector register to memory	For storing data in memory e.g. plaintext
vluxei8.v	Loads 8 bit to a vector register using a base address and a vector containing offsets to be added.	Used for S-Box lookup, shifting, rotation during Key Expansion
vor.vv	Performs bitwise OR between two vector elements	Used to copy elements from one vector to another
vredxor.vs	Performs XOR between all elements in vectors, reducing them to a scalar result	Used in MixColumns to XOR all results needed to find the new column element
vmv.v.x	Splats a scalar to all active	Used to clear registers, to

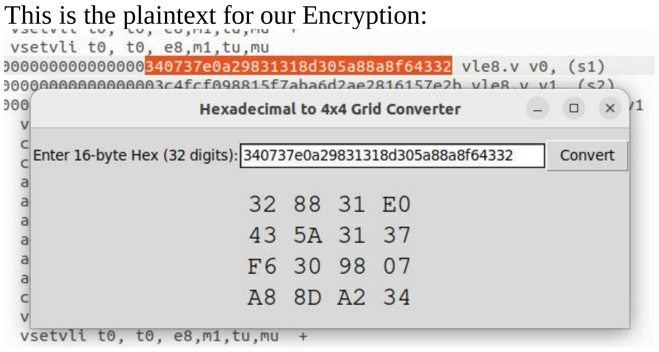
	elements of a register	mask 0x80 and 0x1B across all elements for Galois Multiplication
vmsne.vi	Compares elements of the vector to immediate, sets mask if not equal	Used in Galois Multiplication by 2 for conditional XOR
vsll.vi	Performs a logical left shift on elements by bits specified in immediate	Used in Galois Multiplication by 2
vslideup.vi	Shifts elements upwards by the number of positions specified in the immediate field	Used in Mixed Columns to position the resulting new elements to their correct position

V. Results and Analysis

A. Encryption

We have thoroughly tested our code and produced ciphertexts corresponding to the plaintext. The resulting ciphertext was given as input to our decryption algorithm and we successfully achieved the plaintext. The results are checked in the log.txt file that maintains the data movement. Using a Hexadecimal to 4x4 converter, we have shown outputs for the first round after each step. The results have been verified by an animation available online [3]

This is the plaintext for our Encryption:



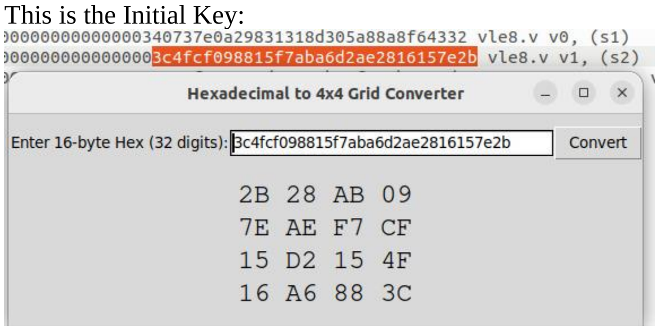
```
vsetvli t0, t0, e8,m1,tu,mu
000000000000000340737e0a29831318d305a88a8f64332 vle8.v v0, (s1)
```

Hexadecimal to 4x4 Grid Converter

```
Enter 16-byte Hex (32 digits): 340737e0a29831318d305a88a8f64332

32 88 31 E0
43 5A 31 37
F6 30 98 07
A8 8D A2 34
```

```
vsetvli t0, t0, e8,m1,tu,mu
```

This is the Initial Key:

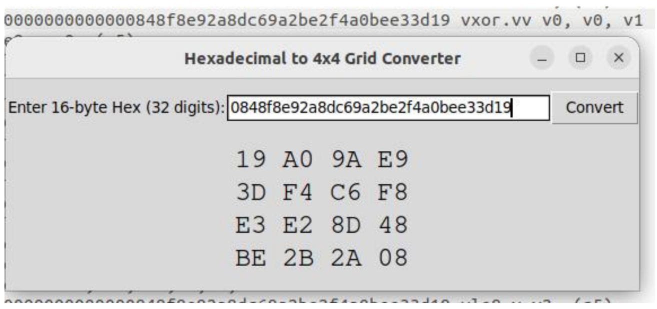


```
000000000000000340737e0a29831318d305a88a8f64332 vle8.v v0, (s1)
0000000000000003c4fcf098815f7aba6d2ae2816157e2b vle8.v v1, (s2)
```

Hexadecimal to 4x4 Grid Converter

```
Enter 16-byte Hex (32 digits): 3c4fcf098815f7aba6d2ae2816157e2b

2B 28 AB 09
7E AE F7 CF
15 D2 15 4F
16 A6 88 3C
```

AddRoundKey Step: (not a part of loop, just for Round 0)

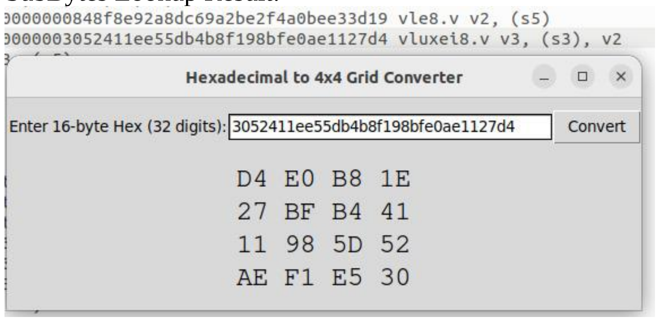


```
0000000000000848f8e92a8dc69a2be2f4a0bee33d19 vxor.vv v0, v0, v1
```

Hexadecimal to 4x4 Grid Converter

```
Enter 16-byte Hex (32 digits): 0848f8e92a8dc69a2be2f4a0bee33d19

19 A0 9A E9
3D F4 C6 F8
E3 E2 8D 48
BE 2B 2A 08
```

SubBytes Lookup Result:

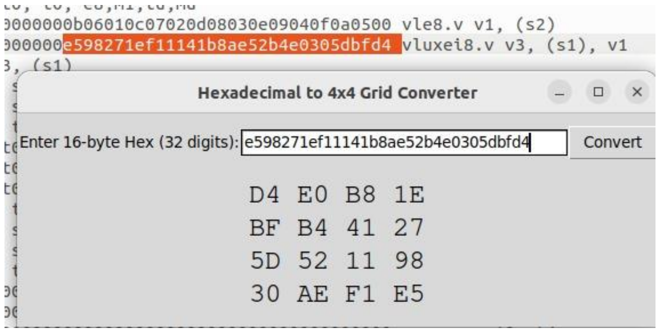


```
0000000848f8e92a8dc69a2be2f4a0bee33d19 vle8.v v2, (s5)
0000003052411ee55db4b8f198bfe0ae1127d4 vluxei8.v v3, (s3), v2
```

Hexadecimal to 4x4 Grid Converter

```
Enter 16-byte Hex (32 digits): 3052411ee55db4b8f198bfe0ae1127d4

D4 E0 B8 1E
27 BF B4 41
11 98 5D 52
AE F1 E5 30
```

ShiftRows Result:

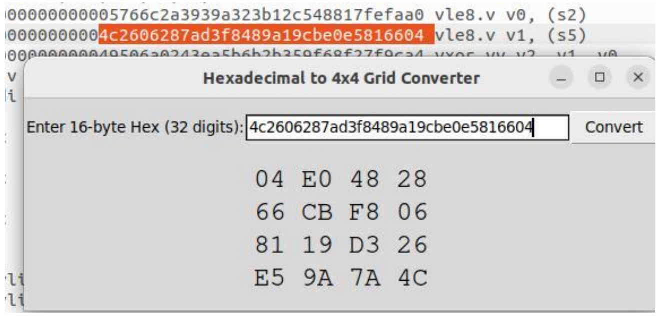


```
0000000b06010c07020d08030e09040f0a0500 vle8.v v1, (s2)
000000e598271ef11141b8ae52b4e0305dbfd4 vluxei8.v v3, (s1), v1
3, (s1)
```

Hexadecimal to 4x4 Grid Converter

```
Enter 16-byte Hex (32 digits): e598271ef11141b8ae52b4e0305dbfd4

D4 E0 B8 1E
BF B4 41 27
5D 52 11 98
30 AE F1 E5
```

MixColumns Result:

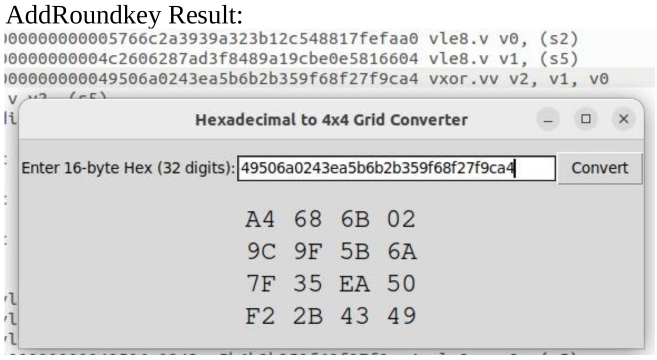


```
00000000005766c2a3939a323b12c548817fefaa0 vle8.v v0, (s2)
00000000004c2606287ad3f8489a19cbe0e5816604 vle8.v v1, (s5)
```

Hexadecimal to 4x4 Grid Converter

```
Enter 16-byte Hex (32 digits): 4c2606287ad3f8489a19cbe0e5816604

04 E0 48 28
66 CB F8 06
81 19 D3 26
E5 9A 7A 4C
```

AddRoundkey Result:



```
00000000005766c2a3939a323b12c548817fefaa0 vle8.v v0, (s2)
00000000004c2606287ad3f8489a19cbe0e5816604 vle8.v v1, (s5)
000000000049506a0243ea5b6b2b359f68f27f9ca4 vxor.vv v2, v1, v0
```

Hexadecimal to 4x4 Grid Converter

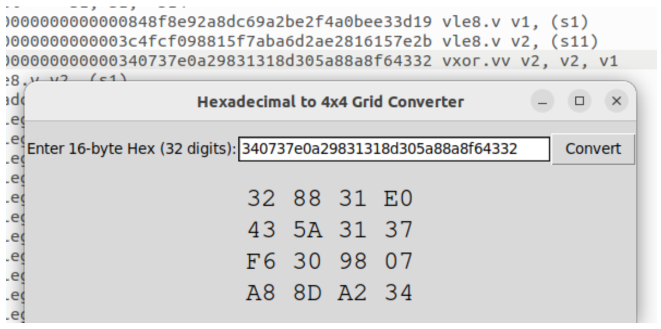
```
Enter 16-byte Hex (32 digits): 49506a0243ea5b6b2b359f68f27f9ca4

A4 68 6B 02
9C 9F 5B 6A
7F 35 EA 50
F2 2B 43 49
```

This is one complete round. The result of the last step then undergoes the the same steps in the following order: SubBytes, ShiftRows, MixCols, AddRoundKey.

B. Decryption

The steps for decryption are the same. As proof, we are using the ciphertext from our encryption result as input. The result of decryption after 11 rounds returns us the plaintext.

Result of Decryption:



```
)00000000000000848f8e92a8dc69a2be2f4a0bee33d19 vle8.v v1, (s1)
)000000000003c4fcf098815f7aba6d2ae2816157e2b vle8.v v2, (s11)
)00000000000340737e0a29831318d305a88a8f64332 vxor.vv v2, v2, v1
```

Hexadecimal to 4x4 Grid Converter

Enter 16-byte Hex (32 digits): 340737e0a29831318d305a88a8f64332

```
32 88 31 E0
43 5A 31 37
F6 30 98 07
A8 8D A2 34
```

C. Challenges Faced

1. Limited Resources Available for Vector-ISA Codes and Documentation

One of the primary challenges faced during the implementation of AES using the Vector-ISA was the scarcity of available resources and documentation. The Vector-ISA for RISC-V is relatively new, and there is a limited amount of existing code, tutorials, and examples to reference. This lack of readily accessible information and community support made it difficult to find solutions to specific problems and required us to invest significant time in understanding and implementing vectorized operations from scratch. Consequently, the development process involved a lot of trial and error, experimentation, and custom optimization to ensure the algorithm's efficient execution. Debugging was not easy as we had to go through the log file and trace for errors.

2. RVV Documentation Issues:

The Vector Unit-Stride Segment Load and Store instructions (vlseg/vsseg) are intended to move packed contiguous segments into multiple destination vector register groups. The prefixes "vlseg" and "vsseg" are used for unit-stride segment loads and stores, respectively. However, there was confusion regarding the interpretation of the "8" in these instructions. The examples given below were not very helpful.

VI. References

[1] “Efficient AES Encryption on Vector Architectures”, vol. 64, no. 9, pp. 2467-2479, Sept. 2015, by O'Connor and Baugh.

[2] “Vectorized AES Encryption for High-Performance and Low-Energy Platforms”, Jun. 2018, by Wang, Zhang, and Li.

[3]https://formaestudio.com/rijndaelinspector/archivos/Rijndael_Animation_v4_eng-html5.html “AES Rijndael Cipher explained as a Flash animation,” a video by AppliedGo.

VII. Acknowledgments

We extend our heartfelt thanks to Dr. Salman Zaffar, our instructor for Computer Architecture and Assembly Language, for his invaluable guidance and support

VIII. Code link

https://github.com/azkaaqeel/AES-128-Implementation-Using-RISC-V-Vector-ISA