Journal of Advances in Science and Technology
Vol. 15, Issue No. 2, September-2018, ISSN 2230-9659

Advanced Encryption Standard instruction Set Extensions for EC Cryptography

Dr. Sridevi*

Department of Computer Science, Karnatak University, Dharwad, Karnataka

Abstract – This paper demonstrate that the case by showing the use of word-level binary polynomial multiplication for acceleration of the Advanced Encryption Standard (AES) algorithm. Considerable fraction of the computation effort of a software implementation of AES is spent in the MixColumns and InvMixColumns transformations. Consequently, these transformations are a worthwhile target for optimization as demonstrated by the approaches of T-table lookup or alternative representation of the AES State. Performance of MixColumns implementations lies in the fact that the required multiplications in the binary extension field GF(28) are not supported by modern processors and need to be emulated by shift and XOR instructions. Instruction set extensions for Elliptic Curve Cryptography (ECC) include support for arithmetic in large binary extension fields. This analyzes how well these custom instructions are suited for accelerating a software implementation of AES on 32-bit platforms. Taking fast AES implementations for 32-bit processors as reference.

Keywords: AES, MixColumns, ECC, XOR, etc.

----------X----------

1. IMPLEMENTING AES ON 32-BIT PROCESSORS

On 32-bit platforms, most of the AES operations can be implemented with table lookups using the T-table approach. A set of T-tables can be used to implement a specific part of the AES algorithm. For each such part, there is a choice between the use of a single table of 256 entries of 32-bit words or a set of four such tables, i.e., a size of 1KB or 4 KB, respectively. The three additional tables in the set of four tables are just rotated versions of the original table. Therefore, a single T-table is sufficient if the necessary rotations are executed at runtime.

The part of AES most worthwhile to be implemented with T-tables is the combination of SubBytes, ShiftRows and MixColumns, which is used in normal encryption rounds. The SubBytes and ShiftRows transformations in the final round can also be implemented with another set of T-tables, but the potential speedup is rather small. Similarly, InvSubBytes, InvShiftRows and InvMixColumns can be realized with T-tables. However, in such a case it is necessary to employ the equivalent inverse cipher structure[1]. InvSubBytes and InvShiftRows in the final decryption round can also be done with T-tables. The use of the equivalent inverse cipher structure necessitates a more complex key expansion, as most of the round keys (except the first and last) must be transformed with InvMixColumns. When a precomputed key schedule is employed, the additional transformations normally pose no problem, as the costly key expansion is only done once per cipher key. However, if on-the-fly key expansion is to be used, AES decryption with T-tables for the rounds can become rather inefficient. The InvMixColumns operation in the key expansion can also be implemented with another set of T-tables.

The minimal size of lookup tables for a software AES implementation (without resorting to bit-slicing techniques) is 256 bytes for SubBytes and InvSub-Bytes, respectively. In principle, the byte substitutions could be calculated on-the-fly through their defining arithmetic operations: Inversion in $GF(2^8)$ and affine transformation. However, this would be very slow on conventional processors, which are not fit for arithmetic in binary extension fields. Therefore, lookup of the S-box remains the only practical solution.

The rest of the AES round transformations can be calculated with reasonable computational effort. SubBytes and InvSubBytes can be combined with ShiftRows and InvShiftRows, respectively, if the bytes are arranged accordingly after substitution. Such a combined operation is possible as SubBytes and Shift-Rows are consecutive operations and their order of execution can be switched arbitrarily. The combination delivers the shifting of the rows at no additional cost. AddRoundKey can be realized with a few XOR instructions, which are found on virtually all microprocessors.

The bit-slicing technique can be applied in order to get AES implementations which do not require any lookup tables. Each AES State is distributed amongst a number of registers where each register contains parts of a number of different States. The transformations themselves are expressed as logical operations.

Multiple AES operations can be performed simultaneously and the average cost per block is comparable to a conventional implementation provided that the word size of the processor is sufficiently large. The absence of data-dependent table lookups makes bit-slicing implementations resistant against cache-based timing attacks. However, the latency for a single AES operation is very large. This is especially a problem in situations where it is not possible to parallelize the AES operations, e.g., in CBC mode encryption. As bit-slicing leads to a less general solution for realizing AES, we will concentrate on conventional implementations.

Hence, depending on the implementation strategy, AES encryption requires between 256 bytes (just one S-box table) and 8KB (two sets of T-tables of 4KB each) of lookup tables. For AES decryption, the range goes from 256 bytes up to a maximum of 12 KB. Depending on the acceptable code size, the T-tables can be statically included in the code section of the program or generated at runtime.

In the first case, the tables reside in the program memory of the processor while in the second case, they are placed in the working memory. The performance of AES implementations with T-tables is highly dependent on the properties of the memory subsystem of the processor. Especially on systems with slow memory and no or minimal cache, it can be faster to calculate the AES round transformations directly. Another important design aspect is the storage of the State on 32-bit architectures. At the beginning of encryption or decryption, the State is filled with the plaintext or ciphertext. Herein, the first four bytes of the input make up the first column of the State, the next four bytes the second column, etc. On 32-bit processors, four bytes are usually packed into a 32-bit word in order to increase utilization of registers and the datapath. A common choice is to hold the four columns of the State in four 32-bit registers. We will denote an AES implementation with such a storage strategy as column-oriented. The well-known AES implementation of Brian Gladman [3] is an example of a column-oriented implementation.

The MixColumns and InvMixColumns operations interpret the State bytes and State columns as elements of binary extension fields and require operations which are normally not supported by common microprocessors. When these transformations are calculated by the processor, the finite field operations must be realized with instructions for logical operations, shifting and integer arithmetic.

Consequently, a considerable part of AES is spent on calculating the MixColumns and InvMixColumns operations. Bertoni et al. have presented an alternate way for calculating MixColumns and its inverse on 32-bit platforms. Their strategy requires that the rows of the State are held in 32-bit words instead of the columns2. The key advantage of this method is the possibility to multiply all four bytes of each word simultaneously with the same constant from $GF(2^8)$ without the need to shift the results into place. Although this strategy requires a transposition of the State matrix at the beginning and end of AES, a transposition of the cipher key and a more complex key expansion, the whole AES operation is commonly faster than a column-oriented implementation. The performance gains are especially significant for decryption, because InvMixColumns is much easier to calculate with the rows of the State than with the columns. The algorithms for calculating MixColumns and InvMixColumns using the State columns and State rows, as well as possible optimizations using ECC instruction set extensions.

2. OPTIMIZING AES USING INSTRUCTION SET EXTENSIONS

MixColumns and InvMixColumns require addition and multiplication of elements of the binary extension field $GF(2^8)$ and of polynomials over $GF(2^8)$. Addition in $GF(2^8)$ is defined as a bitwise XOR. Multiplication in $GF(2^8)$ can be seen as multiplication of binary polynomials (i.e., coefficients mod 2), followed by a reduction with an irreducible polynomial. Arithmetic with polynomials over $GF(2^8)$ follows the conventional rules for polynomials, using addition and multiplication in $GF(2^8)$ for the coefficients.

In the context of Elliptic Curve Cryptography, various instruction set extensions for arithmetic in binary extension fields GF(2m) have been proposed. The word-level multiplication of binary polynomials has been identified as one of the key operations by Ko¸c et al. in, where this operation was denoted as MULGF2. In, a small set of instructions (including one for MULGF2) for the MIPS32 architecture has been presented and their impact on ECC implementations over GF(p) and GF(2m) has been evaluated. We have used three of these instructions to speed up AES implementations. Table1 lists the instruction names used for MIPS32 in and the mnemonics we have used for our SPARC implementation along with a short functional description. We will employ the SPARC names in the following. All three instructions work on a dedicated accumulator whose size must be at least twice the word size, i.e., in our case at least 64 bits.

Table 1: The ECC instruction set extensions used to speed up AES.

SPARC	MIPS32	Description
gf2mul	mulgf2	Multiply two binary polynomials
gf2mac	maddgf2	Same as gf2mul with addition of result to accumulator
shacr	sha	Shift lower word out of accumulator

The instructions gf2mul and gf2mac interpret the two operands as binary polynomials, multiply them, and put the result in the accumulator. They differ in that gf2mul overwrites the previous accumulator value while gf2mac adds the polynomial product to it. The shacr instruction writes the lowest word of the accumulator to a given destination register and shifts the accumulator value to the right by a distance of 32 bits. All timing estimations for code snippets presented in this chapter are based on the following properties of the SPARC V8 architecture:

- No rotate instruction is available in the architecture. Rotation is done with two shifts and an OR/XOR instruction.
- In order to set a constant value with more than 13 bits in a register, two instructions are required.
- There are enough free registers to hold up to three constant words throughout the calculation of MixColumns or InvMixColumns.

3. COLUMN-ORIENTED IMPLEMENTATION

For MixColumns and InvMixColumns, each new column can be calculated separately from the old column. This property is used if the four columns of the State are held in separate 32-bit words. The following code calculates MixColumns for a single State column in a conventional fashion. At the beginning, the input column is held in the variable column and at the end, the transformed column is written into this variable.

```
01 word double, triple; 02 double = GFDOUBLE(column); 03 triple = double ^ column;
04 column = double ^ ROTL(triple, 8) ^ ROTL(column, 16) ^ ROTL(column, 24);
```

Code 1: MixColumns for a single state column (conventional).

The operator ^ denotes bitwise XOR. The function GFDOUBLE interprets the four bytes of column as four elements of GF(2^8) and doubles them individually. The function ROTL rotates the word to the left by the given number of bits. The basic idea behind the code is that each byte of the resulting column consists of a weighted sum of the four bytes of the old column. Multiplication of all four bytes with the GF(2^8) constants 02 and 03 is done in line 2 and 3 and the result is stored in double and triple, respectively. In line 4, the bytes are rotated into the correct positions and summed up.

The function GFDOUBLE requires about 10 instructions. The function ROTL takes between one and three instructions. The actual number depends on whether the processor features a dedicated rotate instruction. As this is not the case for the SPARC V8 architecture, we will consider the cost of ROTL to be three instructions in the following. Logical operations like the XORs in line 4 are considered to map to a single instruction. The calculation of a single column requires one GFDOUBLE, three ROTL and four XOR operations, which results in a total instruction count of 23 for the code in code :1

When the ECC instruction set extensions listed in Table1 are available, it is possible to calculate a column much faster. In order to do this, we use the definition of MixColumns in terms of a polynomial multiplication. More precisely, MixColumns can be described as a multiplication of two polynomials of degree 3 with coefficients in GF(2^8). The input column is interpreted as the first polynomial, whereas the second polynomial is fixed to the value of 03 · t3 + 01 · t2 + 01 · t + 02. The following code calculates MixColumns for a single column.

```
01 word mask, low_word, high_word; 02 mask = column & 0x80808080;          // Extract MSBs
03 mask = mask >> 7;               04 GF2MUL(column, 0x01010302);          // Polynomial mult.
05 GF2MAC(mask, 0x00011a1b);       // Coefficient reduction
06 SHACR(low_word);                // Degrees 0-3 of poly.
07 SHACR(high_word);               // Degrees 4-6 of poly.
08 column = low_word ^ high_word;  // Polynomial reduction
```

Code 2: MixColumns for a single state column (using extensions).

If the instruction set extensions are available, the three functions GF2MUL, GF2MAC, and SHACR directly map down to the corresponding processor instructions. The rest of the code consists of simple logical operations. The main idea behind this code is illustrated in Figure3. There are three phases in the whole calculation:

- Polynomial multiplication
- Reduction of polynomial coefficients
- Polynomial reduction

Line 4 performs the multiplication of the input column with the constant polynomial 01· t3 + 01 · t2 + 03 · t1 + 02. Note that as the bytes of the column represent the polynomial coefficients with ascending degree (i.e., the byte at the word's most significant position is the coefficient of t0), the coefficients of the constant polynomial have been rearranged accordingly to yield a product with ascending coefficient degree. The polynomial multiplication puts the first block of coefficients in Figure3 in the accumulator. Note that the coefficients are displayed

separately, but that they are in fact all contained as additive contributions in the multiplication result.

The coefficients si, which have been multiplied with the value 03 or 02, may no longer be in the reduced form of binary polynomials of degree smaller or equal to 7, but can be of larger degree. More specifically, 3 si or 2 si are no longer in reduced form if (and only if) the most significant bit (MSB) of si is 1. Such values are called residue values and they can be reduced to a degree smaller or equal to 7 by adding the reduction polynomial of the finite field, which is $x^8 + x^4 + x^3 + x + 1$ (0x11b) in the case of AES.

In order to achieve the reduction of the GF(2^8) coefficients, we add a reduction value ri for each coefficient 3si and 2si. In the case that the MSB of si is 1, the according ri is set to 0x11b, otherwise ri is set to 0. The reduction values are shown in the middle of Figure3 with corresponding coefficients and reduction values marked in the same color. After the addition of the reduction values to the result of the polynomial multiplication, all coefficients are fully reduced. The reduction values ri can be calculated by extraction of the corresponding MSBs of the coefficients si (lines 2 and 3) and the multiplication of these values with the constant 0x00011a1b (line 5). This constant is the sum of 0x11b aligned to the two lower bytes of the word, and the multiplication generates the reduction values ri in the required positions. In line 5, the reduction values are also added to the previous multiplication result in the accumulator by the GF2MAC operation.

In line 6 and 7, the polynomial is read into two variables and in line 8 the reduction of the polynomial is performed. Due to the special nature of the reduction polynomial $p(t) = t^4 + 1$, the coefficients for degrees 4 to 6 must be added to the coefficients of degree 0 to 2, respectively (the coefficient for degree 3 stays unchanged). This is easily done by an XOR of the low and the high word of the accumulator.

The calculation of a single column for MixColumns shown in code1 therefore requires thirteen instructions. This includes the generation of the three constant values which are used in the process (0x80808080, 0x01010302, and 0x00011a1b). But as these values only need to be generated once per MixColumns, there is an average number of 8.5 instructions required to calculate a single column3.

The optimizations for InvMixColumns work in a similar fashion, with the exception that the reduction values ri are generated with an additional GF2MUL operation by performing the polynomial multiplication with the highest three bits of each coefficient alone. The according code is shown in code 4. It takes approximately 16 instructions to calculate one column, which is much faster than the conventional approach.

Lines 2 to 10 generate the reduction bits and put them at the correct position in a single word. In line 12, these reduction bits are used to perform the coefficient reduction on the result of line 11. The rest of the code is similar to the one for MixColumns.

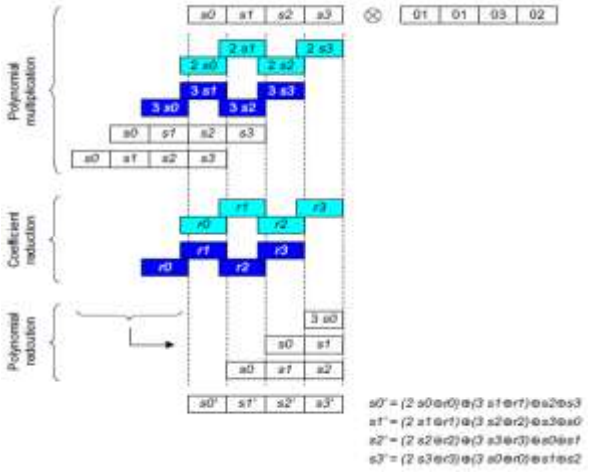

Figure 3: Polynomial multiplication and reduction to yield a column after MixColumns.

4. ROW-ORIENTED IMPLEMENTATION

In a row-oriented AES implementation the MixColumns and InvMixColumns operations are calculated together for the complete State. The strength of this method lies in the possibility to reuse intermediate results for all four columns of the State. This advantage is especially significant in the relatively complex InvMixColumns operation. The conventional row-oriented MixColumns uses four GFDOUBLE operations, while InvMixColumns requires seven. Code 5 depicts the code for a conventional implementation of the GFDOUBLE operation. Here, the reduction information is extracted from poly in lines 2 to 4. The actual doubling of the four bytes takes place in line 5. Afterwards, the reduction is performed (line 6). This version of GFDOUBLE requires 10 instructions, but the reuse of the bitmasks in consecutive GFDOUBLE operations leads to a lower instruction count. There are four consecutive doublings in both MixColumns and InvMixColumns, which can be done in an average of 7 instructions each when the bitmasks are reused.

```
01 word mask, low_word, high_word; 02 mask = column & 0xE0E0E0E0;      // Get highest 3 bits
03 GF2MUL(mask, 0x090D0B0E);      // Mask poly. mult.  04 SHACR(low_word);
05 SHACR(high_word); 06 low_word = low_word ^ high_word;      // Mask poly. red.
07 high_word = high_word << 24; // Shift mask  08 low_word = low_word >> 8; // Shift mask
09 mask = low_word ^ high_word; // Assemble mask 10 mask = mask & 0x07070707;    // Select red.
Bits 11 GF2MAC(column, 0x090D0B0E);      // Polynomial mult.
12 GF2MAC(mask, 0x11b);                  // Coefficient red.
13 SHACR(low_word);                      // Degrees 0-3 of poly.
14 SHACR(high_word);                     // Degrees 4-6 of poly.
15 column = low_word ^ high_word;        // Polynomial reduction
```

Code 4: InvMixColumns for a single state column (using extensions).

```
01 word mask; 02 mask = poly & 0x80808080; 03 mask = mask >> 7;
04 mask = mask * 0x1b // reduction mask 05 poly = (poly & 0x7f7f7f7f) << 1;
06 poly = poly ^ mask;
```

Code 5: GFDOUBLE for row-wise MixColumns/InvMixColumns (conventional).

With the help of the gf2mul, gf2mac and shacr instructions, the GFDOUBLE operation can be done slightly faster than with conventional instructions. Such an optimized version of GFDOUBLE is shown in code 6.The reduction information is extracted in a similar manner, but doubling and reduction are done with GF2MUL and GF2MAC, respectively. The optimized version takes about 7 instructions. When reusing the bitmask in four consecutive doublings, the average instruction count goes down to approximately 5.

```
01 word mask; 02 mask = poly & 0x80808080; 03 mask = mask >> 7;          // reduction mask
04 GF2MUL(poly, 0x2); 05 GF2MAC(mask, 0x11b);          // GF(2^8) coefficient reduction
06 SHACR(poly);
```

Code .6: GFDOUBLE for row-wise MixColumns/InvMixColumns (using extensions).

5. RESULTS

In order to implement and evaluate our new AES implementation approaches, we used a version of the LEON2-CIS which includes the required ECC extensions. The processor configuration includes a (32×32)-bit unified integer/polynomial multiply-accumulate unit with a 72-bit accumulator (including 8 guard bits for integer multiply-accumulate). At the heart of this multiply-accumulate unit is a (32×16)-bit unified integer/polynomial multiplier. The processor has been implemented on the GR-PCI-XC2V FPGA board.

On this version of the LEON2-CIS, a gf2mul instruction executes in three cycles, while a gf2mac instruction takes only a single cycle6. The shacr instruction always finishes in one cycle. In order to estimate the hardware cost of the extensions, we have compared the synthesis results of a "conventional" LEON2 featuring a conventional (32×16)-bit integer multiplier to our LEON2-CIS variant. The latter requires about 5,5 kGates more than the reference version, whereby the added functionality encompasses not only all the extensions from], but also a signed multiply accumulate instruction. Unfortunately, the LEON2 does not offer a configuration with a (32×16)-bit multiply-accumulate unit, so the sole cost of the instructions for binary polynomials cannot be determined easily.

We have made tests with Gladman's AES code [3] using it both as reference as well as an instance of a column-oriented implementation. However, Gladman's code only allows to optimize the calculation of a single column and not of the whole MixColumns transformation. Therefore, we have implemented our own version of a column-oriented AES which is more easily optimized. Furthermore, we have implemented our own version of a row-oriented AES following the ideas from. Our column-oriented and row-oriented versions are written in C and support both encryption and decryption both for a pre computed key schedule as well as for on-the-fly key expansion. Moreover, all versions feature a conventional implementation with native SPARC V8 instructions and an optimized implementation where MixColumns and InvMixColumns make use of the ECC instruction set extensions.

Timing measurements have been done using the integrated cycle counter of the LEON2-CIS. The code which performs the measurements has been derived from Gladman's code. In order to get a fair comparison of the different implementation options, we have used a processor configuration with a very large instruction and data cache (4 sets with 16KB each, organized in lines of 8 words). The results therefore reflect performance in an environment with fast memory access or with "perfect" cache.

5.1 Precomputed Key Schedule

Table2: Execution times of AES-128 encryption, decryption and key expansion.

Implementation	Key expansion cycles	Encryption Cycles	Decryption cycles
Gladman NOTABLES	522	1860	3125
Column-Oriented	497	1672	2962
Row-Oriented	738	1636	1954
Gladman NOTABLES optimised	522	1755	1906
Column-Oriented optimised	497	1257	1576
Row-Oriented optimised	738	1502	1567
Speedup		23.1%	19.8%

Table2 lists the timing results for AES-128 encryption and decryption when using a precomputed key schedule. The time for doing the key expansion is also stated. The speedup is calculated between the best conventional implementation and the best optimized implementation (best performance marked in bold). The row-oriented AES is best for both conventional encryption and decryption. For the optimized variants, the column-oriented implementation is best for encryption, while the performance for decryption is nearly identical for the column-oriented and row-oriented version.

5.2 On-the-fly Key Expansion

The timing results in Table3 refer to AES-128 encryption and decryption with on-the-fly key expansion. As Gladman's code does not support this mode, only the results for our column-oriented and row-oriented version are stated. Note that the last round key is supplied to the decryption routine, so that it does not have to perform the whole key expansion at the beginning of decryption. For conventional encryption, the column-oriented AES is

slightly better, while for decryption, the row-oriented version is fastest. For the versions which use the ECC extensions, the column-oriented AES is better for both encryption and decryption. The speedup is again calculated considering the best conventional and optimized version.

Table 3: Execution times of AES-128 encryption and decryption with on-the-fly key expansion.

Implementation	Encryption Cycles	Decryption cycles
Column-Oriented	**2254**	3357
Row-Oriented	2328	**2433**
Column-Oriented optimised	**1674**	**2018**
Row-Oriented optimised	2230	2176
Speedup	25.7%	17.0%

5.3 Code Size and Side-Channel Attacks

The code size for the implementations ranges between 2.5KB and 3.5 KB, where the optimized variants are always smaller than the non-optimized ones. Note however, that the implementations have been optimized for speed and not for code size. Savings through optimization go up to 15% (for column-wise decryption with precomputed key schedule).

The susceptibility to side-channel attacks is not changed through the use of the instruction set extensions. It is therefore necessary to integrate countermeasures into a system which calculates AES using the presented methods, if resistance against side-channel attacks is required.

6. CONCLUSION

This paper demonstrated the use of instruction set extensions originally designed for elliptic curve cryptography for the acceleration of software implementations of AES. Although not specifically designed for that purpose, the use of the three instructions gf2mul, gf2mac and shacr allows performance gains of up to 25%. This speedup can be considered as “free” on processors which already feature these instructions. Generally, the column-oriented AES implementations can be optimized very well with the instruction set extensions.

REFERENCES

[1] J. Daemen and V. Rijmen. The Design of Rijndael. Information Security and Cryptography. Springer, 2002. ISBN 3-540-42580-2.

[2] A. J. Elbirt. Fast and Efficient Implementation of AES via Instruction Set Extensions. In Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops (AINAW 2007), volume 1, pages 396–403. IEEE Computer Society, May 2007.

[3] B. Gladman. Implementations of AES (Rijndael) in C/C++ and Assembler. Available online at http://fp.gladman.plus.com/cryptography_ technology/rijndael/index.htm.

Corresponding Author

Dr. Sridevi*

Department of Computer Science, Karnatak University, Dharwad, Karnataka