

White Paper

A study of AES and its efficient implementation on eSi-RISC

Introduction

This White Paper provides some practical examples of calculating the Advanced Encryption Standard (AES) on 16 and 32-bit versions of eSi-RISC. The basic software implementation is refined using known techniques in the literature, and a novel implementation of Bertoni's transposed MixColumns() transformation provides the most optimised fully software implementation. The final cycle count on eSi-3250 is shown to be better than ARM7TDMI, ARM9TDMI and LEON-2 embedded processor benchmarks and the code density is superior. Finally the white paper looks at how the user-defined instruction extensions can provide additional saving in power, memory and computation cycles.

Embedded systems have become ubiquitous in recent years stemming from the exponential growth in mobile phones, PDAs, portable multimedia devices and smart cards. This has led to a need for strong cryptography to protect users identity, transactions and allow secure billing. This includes security in both wireless communications and authentication. Since embedded systems have limited resources then it is essential that the cryptography overhead is as small as possible.

The main drawback with block ciphers like AES [1] is that they are quite costly to implement in software, but have simple hardware realizations using logical bit operations and manipulation. Offloading these operations from software to hardware using user-defined instructions tightly coupled to a processor leads to considerable clock cycle savings. The AES algorithm is specified in many wireless standards as the MAC protocol encryption method [2]...[8], and also in RFID tags [9].

AES algorithm description

The AES algorithm is described in [1] in terms of operations on a 128-bit block of data (16 bytes), using keys of length 128, 192 and 256 bits. The 16 input bytes are first arranged in a 4x4 State matrix filled columnwise. The encoding operation is described by 4 transformations on the State matrix; SubBytes(), ShiftRows(), MixColumns() and AddRoundKey(). These functions are called sequentially in a number of Rounds between 10 and 14 for the various key lengths. The final round is slightly different from the others in omitting MixColumns(). The code detailed here was adapted from [9].

```
void Cipher(int Nr, int Nk, unsigned char *RoundKey, const unsigned char
*in, unsigned char *out)
{
    int c, r, round = 0;
    unsigned char state[4][4];

    // Copy the input PlainText to state array.
    for (c = 0; c < 4; c++)
    {
        for (r = 0; r < 4; r++)
        {
            state[c][r] = in[c * 4 + r];
        }
    }
}
```

```

// Add the First round key to the state before starting round 1.
AddRoundKey(round, RoundKey, state);

// There will be Nr-1 identical rounds.
for (round = 1; round < Nr; round++)
{
    SubBytes(state);
    ShiftRows(state);
    MixColumns(state);
    AddRoundKey(round, RoundKey, state);
}

// The last round excludes MixColumns().
SubBytes(state);
ShiftRows(state);
AddRoundKey(Nr, RoundKey, state);

// Copy the state array to output array.
for (c = 0; c < 4; c++)
{
    for (r = 0; r < 4; r++)
    {
        out[c * 4 + r] = state[c][r];
    }
}
}

```

Figure 1: Main encryption software routine

The SubBytes() transformation is a non-linear byte substitution that operates independently on each byte of the State using a 256 byte substitution table called an SBOX. The transformation is described by a look-up table operating on each byte and substituting back in-place in the State matrix. For various reasons that will become clear later the state is stored state[col][row].

```

inline static void SubBytes(unsigned char state[4][4])
{
    int c, r;

    for (c=0; c<4; c++)
    {
        for (r=0; r<4; r++)
        {
            state[c][r] = sbox[state[c][r]];
        }
    }
}

```

Figure 2: SubBytes subroutine

In the ShiftRows() transformation, the bytes in the last three rows of State are cyclically shifted left over 1 to 3 bytes respectively. The first row is left un-shifted.

```

inline static void ShiftRows(unsigned char state[4][4])
{
    unsigned char temp;

    // Rotate first row 1 columns to left
    temp = state[0][1];
    state[0][1] = state[1][1];    state[1][1] = state[2][1];
    state[2][1] = state[3][1];    state[3][1] = temp;

    // Rotate second row 2 columns to left
    temp = state[0][2];
    state[0][2] = state[2][2];    state[2][2] = temp;

    temp = state[1][2];
    state[1][2] = state[3][2];    state[3][2] = temp;

    // Rotate third row 3 columns to left
    temp = state[0][3];
    state[0][3] = state[3][3];    state[3][3] = state[2][3];
    state[2][3] = state[1][3];    state[1][3] = temp;
}

```

Figure 3: ShiftRows subroutine

The MixColumns() transformation operates on State column-by-column, treating each column as a four term polynomial. The polynomials over $GF(2^8)$ are multiplied modulo x^4+1 with a fixed polynomial and written back to State in-place. The polynomial multiplication by {02} is carried out with the macro xtime() as described in [1], and multiplication by other polynomials is performed by adding intermediate results.

```

// multiply one byte in GF(2^8) by {02}
#define BPOLY    0x1b
#define xtime(x)  (((x)<<1) ^ (((x)>>7) & 1) * BPOLY)

inline static void MixColumns(unsigned char state[4][4])
{
    int c;
    unsigned char temp, t;

    for (c = 0; c < 4; c++)
    {
        t = state[c][0];
        temp = state[c][0] ^ state[c][1] ^ state[c][2] ^ state[c][3];
        state[c][0] ^= xtime(state[c][0] ^ state[c][1]) ^ temp;
        state[c][1] ^= xtime(state[c][1] ^ state[c][2]) ^ temp;
        state[c][2] ^= xtime(state[c][2] ^ state[c][3]) ^ temp;
        state[c][3] ^= xtime(state[c][3] ^ t) ^ temp;
    }
}

```

Figure 4: MixColumns subroutine

Finally the AddRoundKey() transformation operates columnwise XOR'ing the column with bytes from the Key Schedule, based on the round and column index.

```

inline static void AddRoundKey(int round, unsigned char *RoundKey,
                               unsigned char state[4][4])
{
    int c, r;

    for (c = 0; c < 4; c++)
    {
        for (r = 0; r < 4; r++)
        {
            state[c][r] ^= RoundKey[round * 4 * Nb + c * Nb + r];
        }
    }
}

```

Figure 5: AddRoundKey subroutine

Now the basic algorithm has been introduced it is necessary to get a feel for how this performs on an embedded processor.

The eSi-RISC tool chain is based on the GNU suite and benefits from seamless integration within the Eclipse Integrated Development Environment (IDE). For clarity the algorithms are implemented in “C” using efficient loops and compiled for a Release build with `-O2` optimization which gives a good balance between code size and execution speed. The eSi-RISC is configured with 32-bit wide instruction memory, so that 32-bit instructions can be fetched in a single cycle. The eSi-RISC compiler is very efficient at producing optimised code, often as good as hand crafted assembly language. In addition the instruction set is optimised so that many instructions are encoded in 16-bits leading to a high code density.

The eSi-RISC family of processors have a configurable instruction set, and the following instructions are required for best performance in cryptographic applications.

- barrel shift instruction
- hardware loop
- scaled update addressing

In Table 1 we detail the cycle count and program code size of the basic un-optimised implementation, denoted *aes0*, for the 3 eSi-RISC family members. An additional 256 bytes is needed for SBOX table look-up.

	eSi-1600		eSi-3200		eSi-3250	
	cycles	p codesize	cycles	p codesize	cycles	p codesize
aes0	4748	624	4818	732	4606	618

Table 1: Basic un-optimised AES encryption code.

The eSi-1600 is a 16-bit processor, while eSi-3200 and eSi-3250 are 32-bit, where eSi-3250 has more internal registers than eSi-3200. In the next section we consider how the basic algorithm is restructured in C for faster implementation.

Fast and efficient software enhancement

For pure software enhancement the methods found in [11], [12] and [13] for Intel StongARM SA-1110, Ultra SPARC III, PowerPC, AMD Athlon 64, and Pentium are helpful, but are targeted at high end CPUs that have large and efficient cache memory architectures. For implementations on

embedded processors the references [13], [14] and [15] cover the concepts, while [15] has cycle counts for ARM7TDMI, ARM9TDMI and [16] has cycle counts and code size for LEON-2.

The fastest implementations use T-tables [11], [12] and [13] which require 16 look-ups into large tables to aggregate the all the steps except for AddRoundKey(). These tables are 4kB, and separate tables are required for the last round and for encryption and decryption. This leads to 16kB of memory to achieve the lowest cycle counts. T-table methods compute an AES-128 in 160 to 480 cycles, depending on the processor instruction set, equivalent to 10 to 30 cycles per byte. On processors with a cache this performance can only be realised with a cache size of 8kB or more, and has an impact on other software tasks, whereby the cache is emptied for AES and needs refilling for other tasks to continue efficiently. Apart from excessive memory usage, security is compromised by the threat of cache-based side-channel attacks [17]. Since memory accesses are normally the most energy-intensive instructions then cache hungry algorithms should be avoided. As eSi-RISC is a customised processor it can be configured without a cache and all the memory internal. This gives rise to a simple ROM table approach where the processor is being used mainly as an encryption co-processor in a larger SoC, and the full efficiency of T-tables can be realised.

For a typical embedded processor application where encryption is only one of the tasks being performed, then it becomes necessary to minimise the code-size. The SBOX table is only 256 bytes and is the only table necessary for encryption and decryption. In the subsequent analysis we optimise the code to only use this table and include specific enhancements for a 32-bit processor (although the code will still execute on a 16-bit processor without modification).

There are 3 nested for-loops; assigning the plain-text to the State, AddRoundKey() and assigning the State to the output array that can be performed 4-bytes at a time. By defining a pointer that can index State as four 32-bit words then the new code achieves some reduction in memory accesses. The RoundKey must also be indexed in the same way. The new code fragments are shown below.

```
inline static void AddRoundKey(int round, unsigned char *RoundKey,
unsigned long *state32)
{
    unsigned long *key32 = (unsigned long *) &RoundKey[4*round*Nb];

    // Update state with results
    state32[0] = state32[0] ^ key32[0];
    state32[1] = state32[1] ^ key32[1];
    state32[2] = state32[2] ^ key32[2];
    state32[3] = state32[3] ^ key32[3];
}
```

Figure 6: AddRoundKey subroutine optimised for accessing State by 32-bit columns

```

void Cipher(int Nr, int Nk, unsigned char *RoundKey, const unsigned char
*in, unsigned char *out)
{
    int round = 0;
    unsigned long *in32 = (unsigned long *) in;
    unsigned long *out32 = (unsigned long *) out;
    unsigned char state[4][4];
    unsigned long *state32 = (unsigned long *) &state[0][0];

    // Copy the input PlainText to state array.
    state32[0] = in32[0];
    state32[1] = in32[1];
    state32[2] = in32[2];
    state32[3] = in32[3];

    /* CODE OMITTED FOR CLARITY */

    // The encryption process is over.
    // Copy the state array to output array.
    out32[0] = state32[0];
    out32[1] = state32[1];
    out32[2] = state32[2];
    out32[3] = state32[3];
}

```

Figure 7: Encryption subroutine optimised for accessing State by 32-bit columns

This modification we call aes1 and the resulting speed-up is considerable

	eSi-1600		eSi-3200		eSi-3250	
	cycles	p codesize	cycles	p codesize	cycles	p codesize
aes0	4748	624	4818	732	4606	618
aes1	3391	776	3143	692	2918	682

Table 2: Cycle count comparison after 32-bit column access optimization.

A profile of the code shows that more than 50% of the cycle count is taken up by MixColumns(). In [13] Gladman introduces a 4-byte at a time enhancement to MixColumns that requires the embedded processor to have a barrel-shift instruction. This code performs 4 GF(2⁸) multiplications in a 32-bit column and accumulates with the other columns after rotation.

```

// multiply four bytes in GF(2^8) by {02} in parallel
#define m1 0x80808080
#define m2 0x7f7f7f7f
#define gf_mulx(x) (((x) & m2) << 1) ^ (((x) & m1) >> 7) * BPOLY))
#define upr(x, n) (((x) << (8 * (n))) | ((x) >> (32 - 8 * (n))))
#define fwd_mcol(x) (g2=gf_mulx(x), g2^upr((x)^g2, 3)^upr((x), 2)^upr((x), 1))

unsigned long g2;
inline static void MixColumns(unsigned long *state32)
{
    state32[0] = fwd_mcol(state32[0]);
    state32[1] = fwd_mcol(state32[1]);
    state32[2] = fwd_mcol(state32[2]);
    state32[3] = fwd_mcol(state32[3]);
}

```

Figure 8: MixColumns optimization to perform 4 byte-wise GF(2⁸) multiplications in a 32-bit word

This modification, called aes2, has a very significant impact on cycle count and instructing GCC to unroll loops gives further improvements for a modest increase in program code size. Note that aes2 is 3x faster than aes0 on 32-bit processors, as shown below.

	eSi-1600		eSi-3200		eSi-3250	
	cycles	p codesize	cycles	p codesize	cycles	p codesize
aes0	4748	624	4818	732	4606	618
aes1	3391	776	3143	692	2918	682
aes2	4422	1698	1602	888	1309	894

Table 3: Cycle counts for final optimization of the standard algorithm

Transposed State matrix

The previous section detailed some software enhancements based on the standard algorithm. To get further savings Bertoni noted in [15] that the rotation operations in MixColumns() could be removed by working with the transposed State matrix. The other transformations however need to take the transposed matrix into account, in particular the AddRoundKey needs to operate rowwise instead of columnwise. Bertoni noted that this transposed form gives about 25% saving in decryption and is 2% worse for encryption.

A consequence of performing the operations rowwise is that ShiftRows() can now be performed with an efficient 32-bit barrel shift as shown below.

```
#define dnr(x,n)    (((x) << (32 - 8 * (n))) | ((x) >> (8 * (n))))

inline static void ShiftRows(unsigned long *state32)
{
    state32[1] = dnr(state32[1], 1);
    state32[2] = dnr(state32[2], 2);
    state32[3] = dnr(state32[3], 3);
}
```

Figure 9: ShiftRows can be performed with a 32-bit barrel shift if state is stored in transposed form

In this section a new form of Bertoni's algorithm is introduced that gives good saving in encryption, because it requires only 2 intermediate variables, 1 fewer XORs and the same number of GF multiplies as Bertoni's method. First the Bertoni code fragment is given below with 16 XOR, 4 GF multiplies and 5 intermediate variables. We note that the y[] array, temp and state32[] array need to be held in registers at the same time for the state update.

```

inline static void MixColumns(unsigned long *state32)
{
    unsigned long y[4], temp;

    y[0] = state32[1] ^ state32[2] ^ state32[3];
    y[1] = state32[0] ^ state32[2] ^ state32[3];
    y[2] = state32[0] ^ state32[1] ^ state32[3];
    y[3] = state32[0] ^ state32[1] ^ state32[2];

    state32[0] = gf_mulx(state32[0]);
    state32[1] = gf_mulx(state32[1]);
    state32[2] = gf_mulx(state32[2]);
    state32[3] = gf_mulx(state32[3]);

    temp = state32[0];
    state32[0] = y[0] ^ state32[0] ^ state32[1];
    state32[1] = y[1] ^ state32[1] ^ state32[2];
    state32[2] = y[2] ^ state32[2] ^ state32[3];
    state32[3] = y[3] ^ state32[3] ^ temp;
}

```

Figure 10: Bertoni transformation of MixColumns.

Next we present the new method which takes advantage of the fact that a variable XOR'd with itself is zero to remove some intermediates.

```

inline static void MixColumns(unsigned long *state32)
{
    unsigned long temp, state32_0_;

    temp = state32[0] ^ state32[1] ^ state32[2] ^ state32[3];
    state32_0_ = state32[0];

    state32[0] ^= gf_mulx(state32[0] ^ state32[1]) ^ temp;
    state32[1] ^= gf_mulx(state32[1] ^ state32[2]) ^ temp;
    state32[2] ^= gf_mulx(state32[2] ^ state32[3]) ^ temp;
    state32[3] ^= gf_mulx(state32[3] ^ state32_0_) ^ temp;
}

```

Figure 11: Optimization of MixColumns with new implementation of Bertoni transformation

This new method called aes3 is compared to the non-transposed form below. Whilst the eSi-3200 doesn't benefit from this change, the eSi-3250 is 5% faster.

	eSi-1600		eSi-3200		eSi-3250	
	cycles	p codesize	cycles	p codesize	cycles	p codesize
aes2	4422	1698	1602	888	1309	894
aes3	3560	1678	1783	1090	1252	978

Table 4: Cycle counts for final transposed form aes3, compared to earlier versions.

To get a feel for how good these results really are we compare them to those of the ARM and LEON-2 processor taking the figures from [15] and [16]

CPU	Version	Encryption
ARM7TDMI	Transposed	1675
	Gladman	1641
ARM9TDMI	Transposed	1384
	Gladman	1374
LEON-2	Transposed	1636
eSi-3250	Transposed (aes3)	1252
	Gladman (aes2)	1309

Table 5: Cycle count comparison with other CPUs.

The eSi-3250 is 10% more efficient than the ARM9TDMI, 31% more efficient than the ARM7TDMI and 30% better than LEON-2 and ARM7TDMI. In terms of code size the only comparison is with [16], and the SBOX table is included for comparison with LEON-2.

CPU	Version	Encryption
LEON-2	Transposed	2168 bytes
eSi-3250	Transposed (aes3)	1234 bytes
	Gladman (aes2)	1150 bytes

Table 6: Code size comparison with other CPUs.

The eSi-3250 requires only 57% of the code space compared to LEON-2, because of its mixed 16/32-bit instruction capability giving exceptional code density.

Enhancement with custom instructions

It is now instructive to see if this loop can be accelerated by a user-defined instruction. On eSi-RISC user-defined instructions are implemented in small hardware accelerators attached to the main processor and able to access the register file for up to 2 input arguments and 1 output argument. For simple logical operations like a square root these correspond to very efficient hardware taking only a few logic gates. The user-defined instructions have a "C" subroutine call prototype for simple integration into the code. The actual assembly language instruction is inlined after the arguments are resolved to avoid the overhead of an actual function call. In addition user-defined instructions can take multiple clock cycles to compute their result if required.

Applying this to AES we can either add custom instructions for speeding up the aes2 or aes3 implementations described above. In [16] they chose to add an instruction to perform 1 byte of SBOX substitution and rotation within a 32-bit word, which means working with the Transposed AES state matrix. This combines the ShiftRows() and SubBytes() transformations into one. We implemented a 4-byte version of this instruction able to perform 4 SBOX substitutions and rotation in a single cycle.

```
inline static void SubBytesShiftRows(unsigned long)
{
    state32[0] = user3(state32[0], 0);
    state32[1] = user3(state32[1], 1);
    state32[2] = user3(state32[2], 2);
    state32[3] = user3(state32[3], 3);
}
```

Figure 12: User defined instruction for accelerating SubBytes and ShiftRows in transposed form

To accelerate MixColumns() the GF multiplication and 3 XORs can be combined, leaving a final XOR in the C code. We refer to this implementation as aes3SM to indicate it includes SubBytes() ShiftRows() and MixColumns() optimization.

```

inline static void MixColumns(unsigned long *state32)
{
    unsigned long temp, state32_0_;

    temp = state32[0] ^ state32[1] ^ state32[2] ^ state32[3];
    state32_0_ = state32[0];

    state32[0] = user4(state32[0], state32[1]) ^ temp;
    state32[1] = user4(state32[1], state32[2]) ^ temp;
    state32[2] = user4(state32[2], state32[3]) ^ temp;
    state32[3] = user4(state32[3], state32_0_) ^ temp;
}

```

Figure 13: User defined instruction for accelerating MixColumns in transposed form.

Similar optimizations can be performed on the untransposed form aes2, and the result of these is referred to as aes2SM. The cycle counts for these optimizations are summarised below. The cycle counts for the transposed form are comparable to the T-table look-up methods. The results from [16], where the sbx instruction and GF multiply were accelerated are included here for comparison.

CPU	Version	Encryption	
		cycles	codesize
eSi-3200	aes2SM	833	496
	aes3SM	551	502
eSi-3250	aes2SM	745	508
	aes3SM	488	546
LEON-2	sbx + gf2mul instr (optimized)	612	680

Table 7: Cycle counts and codesize for hardware accelerated CPUs

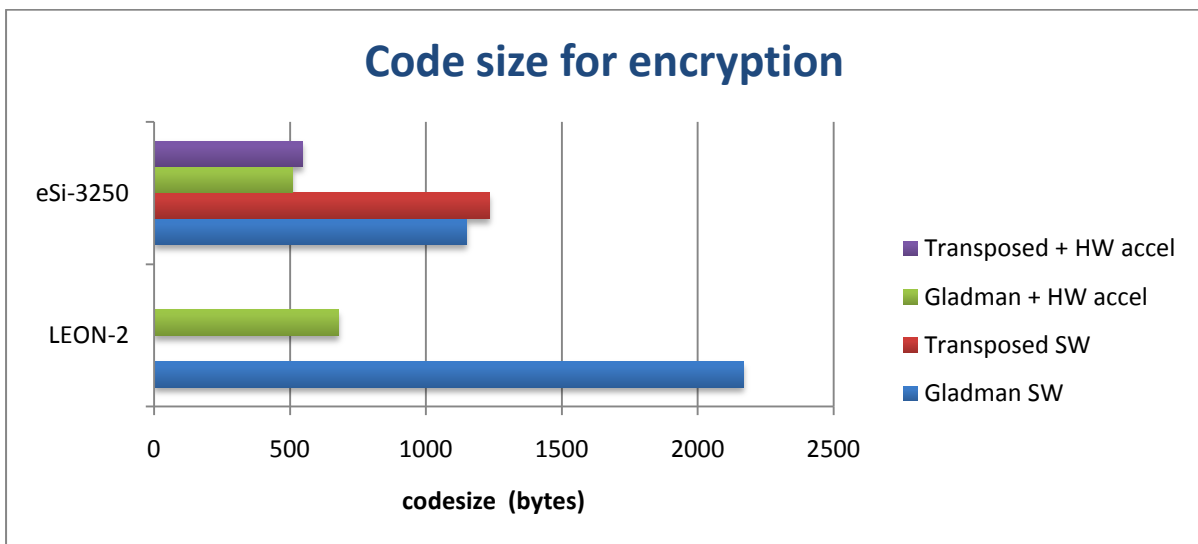
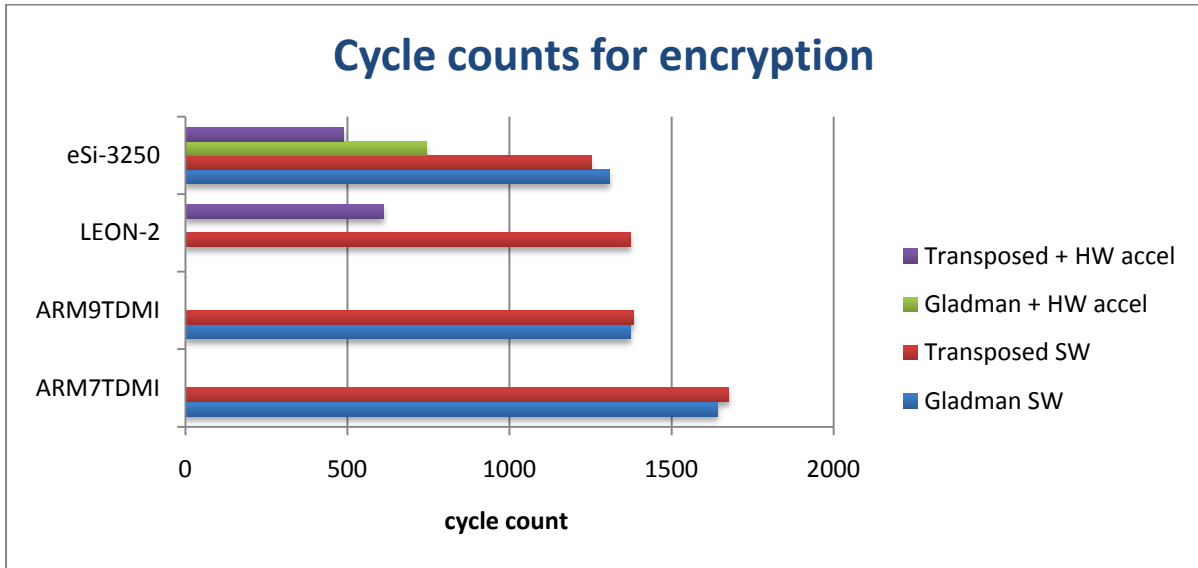
The eSi-1600 cannot be accelerated with the same user defined instructions since it only takes 16-bit words at a time, however it is possible to define instructions to speed AES up on 16-bit machines, but that is beyond the scope of this white paper.

In conclusion the transposed form gives the lowest cycle count when combined with custom instructions, and this is comparable to T-table look-up forms. A speed-up of 2.56x is obtained by the addition of custom instructions compared to the best software implementation. In addition the SBOX table is no longer required in the software, and the same custom instruction can be used to accelerate the Key expansion. Typical full custom hardware AES would perform the transformation in around 40 cycles or 12x faster but without the flexibility offered by the software version.

Conclusion

The AES a ubiquitous algorithm finding application in many embedded systems. This white paper has examined the AES algorithm and gone through the known optimizations to achieve the highest software performance on a 32-bit architecture. The eSi-RISC architecture is well suited to performing this function efficiently using its standard instruction set. It was shown that a novel modification to Bertoni's transposed MixColumns() resulted in eSi-3250 being more efficient than the ARM9TDMI in terms of cycle count. In addition code density was much better than LEON-2 because of the mixed 16/32-bit instruction set. To achieve higher performance the eSi-RISC user-

defined instructions were explored as a simple expansive mechanism for offloading computation. The results are summarised in the charts below.



The application code is documented in this paper for reference. A 2.56x cycle count speed up is achieved for a small addition of hardware, and the encryption becomes comparable to using T-tables.

Although this white paper has concentrated on AES encryption the same principles apply to the key schedule generation and decryption, and eSi-RISC provides a efficient solution for these also.

The clear message is that exploiting the software algorithm and executing a subroutine in fewer cycles leads to a lower power consumption, or the ability to trade those cycles for performance in another part of the system.

Many simple algorithms can be accelerated with the addition of small hardware modules. By adding these in as custom instructions the module becomes tightly coupled with the processor and has a bit exact C code simulation model. Examples of suitable algorithms include

- CRC
- IP checksum calculation
- Max/Min search
- Sorting
- Sqrt
- CORDIC
- Data scrambler
- Convolutional encoder
- Viterbi and Turbo decoder
- Sin/cosine generation
- Galois field multiplier
- G.711 u/A-law conversion
- MP3 sample dequantization
- Y'CrCb to R'G'B' conversion

About EnSilica

EnSilica is an established company with many years experience providing high quality IC design services to customers undertaking FPGA and ASIC designs. We have an impressive record of success working across many market segments with particular expertise in multimedia and communication applications. Our customers range from start-ups to blue-chip companies. EnSilica can provide the full range of front-end IC design services, from System Level Design, RTL coding and Verification through to either a FPGA device or the physical design interface (synthesis, STA and DFT) for ASIC designs. EnSilica also offer a portfolio of IP, including a highly configurable 16/32 bit embedded processor called eSi-RISC and the eSi-Comms range of communications IP.

References

1. National Institute of Standards and Technology (NIST). Advanced Encryption Standard (AES). Federal Information Processing Standards (FIPS) Publication 197, Nov. 2001.
2. IEEE Std 802.11-2007, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications
3. IEEE Std 802.15.3-2003, Part 15.3: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for High Rate Wireless Personal Area Networks (WPANs)
4. IEEE Std 802.15.4-2006, Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)
5. IEEE Std 802.16-2009, Part 16: Air Interface for Broadband Wireless Access Systems
6. IEEE Std 802.20-2008, Part 20: Air Interface for Mobile Broadband Wireless Access Systems Supporting Vehicular Mobility— Physical and Media Access Control Layer Specification
7. IEEE Std 1675, Standard for Broadband over Power Line Networks: Medium Access Control and Physical Layer Specifications
8. Homeplug AV White Paper – HomePlug powerline alliance, 2005
9. <http://www.hoozi.com/Articles/AESEncryption.htm>
10. M. Feldhofer, S. Dominikus, and J. Wolkerstorfer. Strong authentication for RFID systems using the AES algorithm. In Cryptographic Hardware and Embedded Systems — CHES 2004, vol. 3156 of Lecture Notes in Computer Science, pp. 357–370. Springer Verlag, 2004.
11. K. Atasu, L. Breveglieri, M. Macchetti. Efficient AES Implementations for ARM Based Platforms. SAC'04, March 14-17, 2004, Nicosia, Cypress.
12. D. Bernstein, P. Schwabe. New AES software speed records. Lecture Notes in Computer Science, Volume 5365/2008, Progress in Cryptology – INDOCRYPT 2008. pages 322-336. SpringerLink.

13. B. Gladman. A Specification for Rijndael, the AES Algorithm. Available at http://www.gladman.me.uk/cryptography_technology, May 2002.
14. J. Daemen, V Rijmen. The Design of Rijndael. Springer-Verlag 2002.
15. G. Bertoni, L. Breveglieri, P. Fragneto, M. Macchetti, and S. Marchesin. Efficient Software Implementation of AES on 32-Bit Platforms. In B. S. K. Jr., Cetin Kaya Koc, and C. Paar, editors, Cryptographic Hardware and Embedded Systems - CHES 2002, volume 2523 of Lecture Notes in Computer Science, pages 159–171. Springer, Berlin, Aug. 2002.
16. S. Tillich, J. Großschadl and A. Szekely. An Instruction Set Extension for Fast and Memory-Efficient AES Implementation. CMS 2005, LNCS 3677, pp. 11-21, 2005.
17. G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero and G. Palermo. AES Power attack based on induced cache miss and countermeasure. Proceedings of the 6th International Conference on Information Technology: Coding and Computing (ITCC 2005), pp. 586-591. IEE Computer Society Press, 2005.