Data-Sliced Implementation of Reed-Solomon and Rijndael on SIMD Processors

Atri Rudra* Vijay Kumar† Pradeep K. Dubey‡ Raghav Bhaskar§
Animesh Sharma¶

Abstract

Galois Field Arithmetic finds wide use in engineering applications, including error-control codes and cryptography. Also, most of the general-purpose processors have recently been enhanced to offer architectural support for SIMD (Single Instruction Multiple Data) processing. Several aspects of Galois Field arithmetic make it suitable for SIMD implementations. This paper explores the use of data-slicing techniques to achieve efficient implementations of Galois Field Arithmetic on general-purpose SIMD architectures. We focus on two representative applications of Galois Field Arithmetic, the Rijndael block cipher and Reed-Solomon error correcting codes. Using these applications, we demonstrate that slicing techniques can achieve significant speedup compared to previously reported implementations. Our techniques require minimal architectural support from any wide data-path SIMD processor: parallel table lookup, bitwise XOR/AND, LOAD/STORE operations. Further, in the case of bit-slicing, only the last four instructions are required. This implies applicability of our approach across wider range of SIMD architectures. We achieve significant performance improvement for both Rijndael encryption, and Reed-Solomon encode/decode. Our implementation of Reed-Solomon encoding achieves more than 20 times throughput compared to any previously published performance figure for general-purpose processors.

*IBM India Research Lab, New Delhi, India (atrirudra@in.ibm.com).

†Amazon.com, Seattle, WA (vijayk@amazon.com). This work was done while the author was at IBM India Research Lab.

‡Broadcom Corporation, CA (pdubey@broadcom.com). This work was done while the author was at IBM India Research Lab.

§Deptt of Mathematics, IIT-Delhi, India.

¶Deptt of Mathematics, IIT-Delhi, India.

1 Introduction

Galois Field Arithmetic finds wide use in engineering applications such as error-control codes [4, 15], cryptography [28], switching theory [2] and digital signal processing [21, 17].

Computationally, Galois Fields are attractive in several ways. All operations are carry-less, the word length is constant and there are no rounding-off issues. Since field elements are assigned k-bit representations for some k, Galois Fields of the form $GF(2^k)$ are popular.

SIMD architectures such as AltiVec extension to PowerPC[8], are employed to obtain high speed implementations in a variety of areas where data parallelism is encountered, such as audio and video compression, image processing, graphics, and signal processing. These architectures which combine wideword processors with architectural extensions designed to enable subword parallelism, also include Intel's MMX [20] and SSE2[7], Sun's VIS [25], and MIPS's MDMX [32].

Several aspects of Galois Field arithmetic make it suitable for SIMD implementations. $GF(2^k)$ arithmetic involves fixed-size operands of k bits each. Often, k is much smaller than the data word width of modern processors (32 or 64 bits), and thus there are potentially large efficiencies to be obtained through parallelism. Further, k is usually a multiple of 8 and often a power of 2, which is convenient from the point of view of subword parallelism. However, as we shall see, there are some complex design obstacles to be overcome before the potential gain can be actualized. In this paper, we present useful techniques to transform Galois Field arithmetic-based computations into efficient SIMD implementations. To this end, we select two representative and perhaps two of the most popular applications of Galois Field arithmetic: Reed-Solomon error correcting code and the Rijndael Block Cipher [6]. Rijndael cipher has been selected by the US National Institue of Standards and Technology(NIST) as the proposed new Advanced Encryption Standard(AES) [33]. All arithmetic operations in Rijndael are in the field $GF(2^8)$ while Reed-Solomon codes also typically have the operations in the same field. For both applications we present efficient SIMD implementations that achieve significant speedup. Our implementations involve taking advantage of subword parallelism by rearranging the underlying computations either through bit-slicing or by using wider, multi-bit slices.

Our slicing-based implementations require only the following instructions to be supported by the target SIMD architecture : parallel table lookup, bitwise XOR and AND, LOAD and STORE operations. Further, in the case of bit-slicing, only the last four instructions are required. This implies good portability across a range of SIMD architectures.

The rest of the paper is organised as follows. In Section 2, we describe the Rijndael Block Cipher and Reed-Solomon codes. Section 3 describes a non-sliced implementation of Rijndael on AltiVec. We discuss some methods of speeding up Galois Field arithmetic computations in Section 4. Finally in Section 5 we describe our sliced implementations of Rijndael and Reed-Solomon and report performance numbers. We conclude in Section 6.

2 Galois Field Arithmetic

A Galois Field is a finite set of elements in which addition, subtraction, multiplication and division (all appropriately defined) can be performed without leaving the set [12]. Addition and multiplication must satisfy the commutative, associative and distributive laws. By $GF(2^n)$, we denote a Galois Field having 2^n elements; where each element is represented by n bits. This n bit representation in Galois Field literature is often regarded as a $n-1$ degree polynomial with coefficients from $GF(2)$ (that is, either 0 or 1). Addition in $GF(2^n)$ is then a simple polynomial addition while $GF(2^n)$ multiplication is polynomial multiplication modulo some "prime" degree n polynomial (which again takes co-efficients from $GF(2)$). $GF(2^n)$ is said to be "constructed" from $GF(2)$ by this "prime" polynomial. Galois Field arithmetic finds wide use in a variety of engineering applications, including error correcting codes and cryptography. A comprehensive exposition of Galois Fields can be found in [12]. In this paper we consider the two applications of Galois Fields arithmetic that are described below.

2.1 Reed-Solomon Error Correcting Codes

Reed-Solomon codes are block-based error correcting codes with a wide range of applications in digital communications and storage. Reed-Solomon codes are used to correct errors in many systems including storage devices, wireless and satellite communications, Digital television, high-speed modems, etc. A Reed-Solomon encoder takes a block of digital data and adds extra "redundant" bits. Errors occur during transmission or storage for a number of reasons (for example noise or interference, scratches on a CD, etc.). The Reed-Solomon decoder processes each block and attempts to correct errors and recover the original data. The number and type of errors that can be corrected depends on the characteristics of the Reed-Solomon code.

For positive integers m ($m \geq 3$) and t ($t < 2^m - 1$), there exists a Reed-Solomon code denoted by $RS(n,k)$. $RS(n,k)$ uses blocks of $n = 2^m - 1$ symbols of m bits each (each symbol is an element of $GF(2^m)$). Each block contains k data symbols and $n-k$ parity symbols, where $(n-k) = 2t$. This code is capable of correcting any combination of t or fewer errors in a block of n symbols, and is called a *t-error-correcting Reed-Solomon code.*

The generator polynomial of $RS(n,k)$ is specified in terms of its roots which are elemnts of the Galois field GF(2^m). Let α be a primitive element[1] in $GF(2^m)$. The generator polynomial $g(x)$ of $RS(n,k)$ is the lowest-degree polynomial over $GF(2)$ which has $\alpha, \alpha^2, \alpha^3, ..., \alpha^{2t}$ as its roots (i.e., $g(\alpha_i) = 0$ for $1 \leq i \leq 2t$). Thus $g(x) = (x-\alpha)(x-\alpha^2)(x-\alpha^3)...(x-\alpha^{2t})$.

A Reed-Solomon encoder appends $2t(= n-k)$ parity symbols to the k information symbols. The $2t$ parity symbols in a systematic Reed-Solomon codeword are given by $i(x) * x^{n-k}$ mod

[1]Note that this means that the elements of $GF(2^m)$ is the set $\{0, 1, \alpha, \alpha^2, \alpha^3, \cdots, \alpha^{2^m-2}\}$.

$g(x)$ where $i(x)$ is the polynomial representing the k input symbols and $g(x)$ is the generator polynomial.

The error correcting procedure is:

1. Compute syndrome $S = (S_1, S_2, .., S_{2t})$ from received polynomial r(X),
2. Determine error-location polynomial $\sigma(X)$ from S.
3. Determine the error-location numbers $\beta_1, \beta_2, ...\beta_v$ by finding the roots of $\sigma(X)$ and correct the errors in r(X).

See [13, 35] for more details. Berlekamp's algorithm is used to find the error-location polynomial.

2.2 Rijndael Block Cipher

Rijndael is a private–key (symmetric key) block cipher which has been selected by the US National Institute of Standards and Technology (NIST) as the Advanced Encryption Standard [33].

Each plain text block (as also the intermediate cipher block) is of 128 bits. The smallest unit in any operation in Rijndael is a byte (which is treated as an element of $GF(2^8)$ where the underlying field polynomial is $x^8 + x^4 + x^3 + x + 1$). All operations are defined on this field. Rijndael is an iterative algorithm, where the number of *rounds* is fixed by the key length. See [6] for the detailed specifications.

3 A Direct Implementation of Rijndael Encryption

We selected AltiVec [8] as our SIMD architecture to implement the Rijndael algorithm. Atlivec has a bank of thirtytwo 128-bit vector registers which means that each block cipher (as well as each intermediate cipher text block or *state* as it is called) can be stored in one vector register. One of the reasons for picking AltiVec is the efficient implementation of 256 byte entry table[2] lookup using the *permute* instruction. In the specifcation of Rijndael, each *state* is looked upon as a 4×4 matrix. The state is asumed to be stored in colomn major form in a vector register.

In this section, a *state* will be denoted as S and each byte will be reffered to as S_{ij} with refrence to the matrix representation of the *state*. Any coloumn in the matrix representation of a *state* will be refered by the vector $\begin{pmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{pmatrix}$ where θ is either of a or b - the former being used as the input of any coloumn transformation and the latter for the output.

[2]Note that such a table can be stored in 16 AltiVec vector registers.

The AltiVec code and its scheduling for one *round* of Rijndael encryption can be found in Appendix A. This is how each step is implemented –

- ByteSub transformation : This transformation is done using a table lookup. Table lookup is done by exploiting the *permute* instruction of AltiVec. The table lookup for 256 entries (each entry being a byte) takes 20 instructions.
- ShiftRow transformation : This transformation is done using a single *permute* instruction.
- MixColoumn transformation : This transformation is the following -

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

  The following observations are used in the implementation.

  – If $x \in GF(2^8)$ then $01 \times x = x$.

  – In $GF(2^8)$ arithmetic $03 = 02 + 01$.

  – If $x \in GF(2^8)$, then $02 \times x$ is a left shift of x followed by a conditional XOR with a constant $GF(2^8)$ number.

- Round Key addition: This step involves exoring of the round key with the *state*.

3.1 Performance

Table 3.1 gives a comparison of our results with other reported results in terms of the number of cycles required per block of Rijndael encryption. Our results are based on performance simulation of AltiVec architecture. By *i-way AltiVec* we denote a superscalar implementation that supports i permute unit(s) (for inter-element SIMD operations) and i simple unit(s) (for intra-element SIMD operations).

4 Techniques for Improvement

Though the implementation of Rijndael outlined in Section 3 achieves a decent performance, it was based on a pretty straightforward implementation of the algorithm using table-lookup support available through *permute* operation in AltiVec and the large register file. Of greater interest would be techniques which have the potential to speed up *any* application based on Galois Field Arithmetic. In this section, we outline two techniques which give superior performance numbers when applied to both Rijndael block cipher and Reed-Solomon error-correcting codes.

Table 1: Cycle counts per block for Rijndael encryption implementations

Worley et al[29]	284 (Pentium)	176 (PA-RISC)	124 (IA-64)
	Requires an 8KB table		
Weiss et al.[26]	210 (Alpha 21264)		
Wollinger et al.[27]	228 (TMS320C6x)		
Aoki et al.[1]	237 (Pentium II)		
Cryptomaniac [30]	90 (external co-processor)		
AltiVec	230 (1-way)	162 (2-way)	
	Requires only a 256 byte table		

4.1 Composite Fields

Composite fields are frequently used in implementations of Galois Field arithmetic [16, 18, 19]. In cases where arithmetic operations rely on table lookups, subfield arithmetic is used to reduce lookup-related costs. This technique has been used to obtain relatively efficient implementations for specific operations such as multiplication, inversion and exponentiation. Much of this work has been aimed at implementation of channel codes. The object has usually been to obtain better software implementations by using smaller tables through subfield arithmetic. Applications to hardware design (such as [18]) have been relatively infrequent.

After [19], the two pairs $\{GF(2^n), Q(y)\}$ and $\{GF((2^n)^m), P(x)\}$ constitute a *composite field* if $GF(2^n)$ is constructed from $GF(2)$ by $Q(y)$ and $GF((2^n)^m)$ is constructed from $GF(2^n)$ by $P(x)$, where $Q(y)$ and $P(x)$ are polynomials of degree n and m respectively. The fields $GF((2^n)^m)$ and $GF(2^k)$, $k = nm$, are isomorphic to each other. Since the complexity of various arithmetic operations differs from one of these fields to the other, we can take advantage of the isomorphism to map a computation from one to the other in search of efficiency. For a given underlying field $GF(2^k)$, our gains depend on the choice of n, m and the polynomials $Q(y)$ and $P(x)$.

This method by itself does not improve upon the Rijndael performance of Section 3. However, in conjunction with the slicing technique described in the next subsection, we realised a more more efficient Rijndael implementation (Section 5).

4.2 Slicing Techniques

Bit-slicing is a popular technique [3] to make use of the inbuilt parallel-processing capability of a wide-word processor. Bit-slicing regards a W-bit processor as a SIMD parallel computer capable of performing W parallel 1-bit operations simultaneously. In this mode, an operand word contains W bits from W different instances of the computation. Initially, W different inputs are taken and arranged so that the first word of the re-arranged input contains the first bit from each of the W inputs, the second word contains the second bit from each input, and so forth. The resulting bit-sliced computation can be regarded as simulating W instances of the hardware circuit for the original computation. Indeed, a bit-sliced computation is designed by first designing a hardware circuit and then simulating it using W-bit registers on the rearranged input described above.

An interesting property of Galois field arithmetic is that circuits implementing any operation in $GF(2^k)$ are solely composed of boolean operations, AND and XOR. These boolean operations are available across all platforms as primitive logical operations and thus, this method can be applied to any general purpose processor and does not require any ISA extension, such as those offered in [5, 24].

A bit-sliced implementation corresponding to an N-gate circuit requires N instructions to carry out W instances of the underlying computation, or N/W instructions per instance. This can be particularly efficient for computations which are not efficiently supported by the target architecture. Consider for instance $GF(2^8)$ multiplication on AltiVec [8]. The straightforward implementation uses three table-lookups and one addition modulo 255. 16-parallel lookups in a 256-entry table can be performed on AltiVec in 20 instructions. Thus, a set of 128 multiplications would require 488 instructions. In comparison, a 137-gate multiplication circuit (done in the composite field $GF((2^4)^2)$) translates into a bit-sliced implementation that can perform 128 multiplications in 137 instructions!

The above computation ignores the cost of ordering the input in bit-sliced fashion and doing the reverse for the output. To evaluate the trade-off correctly that cost must be taken into account. Clearly, that cost will depend on the target instruction set.

However, it is possible to think of scenarios in which a particular operation may be efficiently supported in an architecture. For example, if the AltiVec architecture were to provide an instruction for 16 parallel $GF(2^8)$ multiplications which use the underlying field polynomial of interest to us (a hypothetical but nonetheless technically feasible scenario since the critical path of the multiplication circuit is only six gates deep), then a direct computation would require only eight instructions, compared to the 137 required by the bit-sliced version.

Now consider $GF(2^{16})$ multiplications on this hypothetical version of the AltiVec architecture. It is easy to see that the most efficient computation is neither a direct one, nor a bit-sliced version, but a *byte-sliced* computation, in which each $GF(2^{16})$ multiplication is mapped to a small number of $GF(2^8)$ operations, which are efficiently supported by the architecture in question. In general, the right "slice" to use would depend on the target architecture.

Consider the case when we want to do data-slicing , that is, where the slice is k-bits long. For such a data-slicing to be possible, all primitive operations must have the same input and output size. Galois Field arithmetic is a natural fit to this condition. Note however, that for the special case of bit-slicing (k=1), any arithmetic would satisfy the same input and output size condition (another way to look at this is any n-bit input and m-bit output algorithm can be represented as a boolean circuit of n inputs and m outputs). However, as mentioned earlier, bit-slicing in the case of Galois field has the added advantage of simplicity.

We will try to give an intuitive idea of why data-slicing can give good performance. Let us take the example of bit-slicing with the underlying algorithm being Rijndael– the reader can see that the argument can be easily generalized. In Rijndael, the 128-bit text undergoes some transformations in each step. However, not *all* the 128 bits are involved in each step. So if we have 128 parallel Rijndael operations going on, the same bits (say U in total) in all the 128-bit texts would be unaffected– however, we would still have 128 operations in the straight-forward implementation. In contrast, in the bit-sliced implementation, only (128-U) operations are done that is, bit-slicing is in some sense "fully" parallel.

However, the preformance of data-slicing depends crucially on the choosing the right "slice". As discussed above, some bits in each step do not undergo any transformation– so the "correct" slice depends on how well such "unused" bits get grouped. For example, let us say that there are 8 consecutive unused bits– and we are doing byte-slicing. However, the first four unused bits go in one slice while the rest in another. In this case we would not have any gain from slicing , although better support for byte operations may results in a performance gain (this is the idea of Composite fields). Thus, in general bit-slicing is a "safe" choice.

In the case of SIMD implementations of Galois Field Arithmetic composite field and slicing techniques combine well. To get an intutive understanding of this observation, consider a 128 bit wide architecture and a Galois field computation on $GF(2^8)$ numbers. The 'natural' slicing without going into composite field would be *byte-slicing*. If on the other hand you want to do the computation in $GF((2^4)^2)$, *nibble-slicing* may be a good idea. However, as noted in section 4.2, the correct "slice" finally depends on the target architecure (as was the case in our Rijndael Implemetation , sec 5.1, where *bit-slicing* applied to the Rijndael algorithm in $GF((2^4)^2)$ was more efficient than *nibble-slicing*).

In the next section, we describe our data-sliced implmentations of Rijndael encryption and Reed-Solomon codes and discuss their performace.

5 Data-Sliced Implementations

5.1 Bit-sliced Rijndael Implementataion

We obtained significant speed-up for Rijndael encryption using the bit-slicing. As mentioned in Section 4.2, this implied designing an efficient hardware circuit for the same. The details of this implementation have been reported in [22].

One of the requirements for a data-sliced implementations is the existence of independent streams of plain texts. This till recently was a stringent requirement as encryption for symmetric-key cryptosystems was done in a serial manner (called the Cipher Block Chaining (CBC) mode [31]) for maintaining security. In this mode, the cipher text of one block is XORed with the plain text of the next block before the latter can be encrypted. So one had to analyze applications to extract independent streams– [30] discussed two such scenarios: network traffic experiments and secure disk experiments. However, a new invention in [10] gives a parallel mode of encryption while (provably) maintaining the same security level.

5.1.1 Performance and Analysis

Table 2 lists cycle counts of our bit-sliced implementation of Rijndael Encryption. The numbers apply to any architecture that can support bitwise AND and XOR in addition to LOAD and STORE operations. The three numbers we report correspond to implementations with combined datapath widths for logical operations of 256 bits, 384 bits and 512 bits respectively. Consider for example, the first implementation of AMD's recently announced Hammer[11] architecture (AMD's 64-bit extension of x86 ISA, which includes Intel's 128-bit SIMD SSE-2 multimedia extension of x86). This processor can issue 2 scalar 64b logical operations, along with a 128-bit SIMD logical operation. For the proposed bit-sliced algorithm, this implies an effective datapath width of 256b (with very minor overhead due to separate scalar and SIMD register files). At 2 GHz on such a processor, we can achieve an encryption throughput of 188 MBps or 1.5 Gbps! Moreover, the cycle count goes down with increasing datapath width. A significant performance improvement with respect to can be seen at higher datapath width. For example, an additional SIMD unit with 128b width improves the performance by 30% to 119 cycles/block.

In terms of cycles/block, at logical datapath widths of 384b or more, our implementation offers the best reported throughput for Rijndael encryption on general-purpose processor platforms, such as, x86 or PowerPC, without requiring any cryptography-specific ISA extensions, as in [24, 5]. For software-based VPN/firewall or SSL servers, this can potentially save the cost of any external security coprocessor or crypto accelerator [30, 36]: an external security engine often has a protocol communication overhead with the host server processor.

5.2 Byte-sliced Reed-Solomon Implemetation

5.2.1 Implementation details

We have implemented Reed–Solomon Codes for $m = 8$. This implementation is described below, with some details deferred to Appendix B.

Table 2: Rijndael Encryption Performance (cycles/block) with Bit-Slicing

	Combined Datapath Width for Logical Operations		
Bit-sliced implementation	256b	384b	512b
	170	119	100
	Requires only XOR, AND, L/S, and 2KB table		

- Encoder :

  Typically, a Reed–Solomon Encoder is implemented in a LFSR-fashion [13]. In our byte-sliced version we pack the vector register with the corresponding input bytes of 16 independent streams and simultate the working of 16 parallel Reed–Solomon encoders.

  A pseudo-code of the encoder is presented next in terms of the following primitives:

  1. `LDA (M, b)` : loads the contents of memory location of M into register b.
  2. `LOAD-TABLE (a, b, c)` : Loads the look up table[3] for a into register b and c.
  3. `CONST-MULT (a, b, c, d)` : Multiply b with some n whose look tables are in register c and d and puts the result into a.
  4. `ADD (a, b, c)` : Add contents of registers b and c and stores it in a.

  The following registers are used – I_o contains the i_k's i.e. the input symbols; R_{g1} and R_{g2} are used for loading a Look-up Table; S_i where $i = 0 \cdots (n-k-1)$ are the shift registers for output and $R1$ is a temporary register.

  Restriction on $(n-k)$: $(n-k) \leq 28$[4] i.e. at most 14 errors can be corrected with this design.

  The code:

  $\texttt{LDA}\ (i_{k-1}, I_o)$
  $\texttt{ADD}\ (R1, I_o, S_{n-k-1})$
  $\texttt{LOAD-TABLE}\ (g_{n-k-1}, R_{g1}, R_{g2})$
  $\texttt{CONST-MULT}\ (S_{n-k-1}, R1, R_{g1}, R_{g2})$
  $\texttt{ADD}\ (S_{n-k-1}, S_{n-k-2}, S_{n-k-1})$
  $\texttt{LOAD-TABLE}\ (g_{n-k-2}, R_{g1}, R_{g2})$
  $\texttt{CONST-MULT}\ (S_{n-k-2}, R1, R_{g1}, R_{g2})$
  $\texttt{ADD}\ (S_{n-k-2}, S_{n-k-3}, S_{n-k-2})$

[3] The table has 32 byte entries

[4] This is due to the fact that only thirty two 128-bit registers are available – for higher values of $(n-k)$, register spilling has to be taken care of.

. . .

. . .

CONST-MULT $(S_0, R1, R_{g1}, R_{g2})$

The above code is repeated k times. See Appendix B.1 for the detailed calculation of the throughput for this code in AltiVec.

- Decoder : See Appendix B.2.

5.2.2 Performance

Table 3 compares the throughput of Reed–Solomon (encoding and decoding) implmentations. The performance number of Henrion [9] are for a 133MHz Pentium while that of 4i2i.com [34] is for a 166 MHz Pentium. Our implmentation is for a comparable PowerPC processor implemention of the AltiVec architecture at 133 MHz. As shown in Table 3, byte-sliced implementation offers more than 20 fold speedup over previously published numbers for general-purpose processors.

6 Conclusions

We have presented a data-slicing method which yields efficient implementations of Galois Field Arithmetic on wide dataword SIMD architectures. An interesting feature of this method is the requirement of a small set of SIMD instructions. In particular the bit-sliced implementation requires only simple logical functions along with Load/Store instructions. We achieve significant improvement in the performance of Rijndael encryption and ReedSolomon codecs. In this paper we have combined the previously known technique of bit-slicing with wide dataword SIMD architectures and the application level parallelism of Galois-field based block encryption and error-correction algorithms. This has resulted in significant performance improvement. Directions for future work include exploring the use of composite fields and bit-slicing for Reed–Solomon Error Correcting Codes, and exploiting these techniques for other applications.

References

[1] Kazumaro Aoki and Helger Lipmaa, “Fast Implementations of AES candidates”. In *Proc. Third AES Candidate Conference*, April 13-14, 2000. `http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html`

[2] B.Benjauthrit, I.S.Reed,“ Galois Switching functions and their Applications”. In *IEEE Trans. Comput.*, Vol. C-25, pp.78-86, January 1976.

Table 3: Throughput of Reed-Solomon Implementations in Mbps

	RS(255,251)		RS(255,239)	
Henrion[9]	Encoding			
(133 MHz Pentium)	45		NA	
4i2i.com[34]	Encoding & Decoding		Encoding & Decoding	
(166 MHz Pentium)	12		2.72	
Byte-Sliced AltiVec Implementation	Encoding	Decoding	Encoding	Decoding
(133 MHz PowerPC)	956	332	273	46

[3] Eli Biham, "A Fast New DES Implementation in Software". In *Proc. Fast Software Encryption 4,1997.* `http://www.cs.technion.ac.il/~biham/publications.html`

[4] R.E.Blahut,*Theory and Practice of Error Control Codes*, Reading, MA: Addison-Wesley,1984

[5] J. Burke, J.McDonald and T.Austin, " Architectural Support for Fast Symmetric-Key Cryptography". In *9th International Conference on Architectural Support for Programming Languages and Operating Systems (Asplos-IX)*, 2000.

[6] Joan Daemen and Vincent Rijmen, "AES Proposal: Rijndael". `http://www.esat.kuleuven.ac.be/~rijmen/rijndael/index.html`

[7] Peter Glaskowsky,"Pentium 4 (Partiallly) Previewed", In *Microprocessor report*, Aug. 28,2000.

[8] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung and Hunter Scales, "AltiVec Extension to PowerPC Accelerates Media Processing". In *IEEE Micro*, March-April 2000, pp85-95.

[9] J.C. Henrion ,"An Efficient Software Implementation of a FEC Code". In *Proc. of IDMS'97*, Darmstadt, Germany, September 1997.

[10] Charanjit S. Jutla, "Encryption Modes with Almost Free Message Integrity". In *Proc of Eurocrypt 2001.*

[11] K.Krewell, " AMD Takes Hammer to Itanium". *Microprocessor Report*, Nov. 26, 2001.

[12] Rudolf Lidl and Harald Niederreiter, *Introduction to finite fields and their applications.* Cambridge University Press, Cambridge, Ma., 1986.

[13] Lin and Castello,*Error Control Coding* , Prentice Hall, ISBN: 013283796X, Oct. 1982.

[14] Bradley Lucier, " Cryptography, Finite Fields, and AltiVec". `http://www.AltiVec.org/articles/Crytography_ paper.pdf`

[15] F.J. MacWilliams and N.J.Sloane,*The Theory of Error-Correcting Codes*, Amsterdam:North-Holland 1986.

[16] Edoardo D. Mastrovito, *VLSI Architectures for Computations in Galois Fields.* PhD Thesis, Dept. of EE, Linköping University, Linköping, Sweden 1991.

[17] J.H. McClellan, C.M. Rader,*Number Theory in Digital Signal Processing*, Englewood Cliffs: Prentice Hall, 1979.

[18] Christof Paar and Pedro Soria-Rodriguez, "Fast Arithmetic Architectures for Public-Key Algorithms over Galois Fields $GF((2^n)^m)$". In *Proc. EUROCRYPT '97.*

[19] Chirstof Paar, *Efficient VLSI Architectures for Bit-Parallel Computations in Galois Fields.* PhD Thesis, Institute for Experimental Mathematics, University of Essen, Germany, 1994.

[20] A.Peleg and U.Weiser,"MMX Technology Extension to the Intel Architecture", In *IEEE Micro*, July/Aug. 1996, pp.42-50.

[21] I.S.Reed and T.K.Truong ,"The Use of Finite Fields to Compute Convolutions,"In *IEEE Trans. Infrom. Theory*, vol. IT-21,No.2,pp.208-213, March 1975.

[22] A.Rudra, P.K. Dubey, C.S.Jutla, V.Kumar, J.R.Rao and P.Rohatgi, " Efficient Implementation of Rijndael Encryption with Composite Field Arithmetic". In *Workshop on Cryptographic Hardware and Embedded Systems 2001*, May 13-16,2001.

[23] Bruce Schneier, *Applied Cryptography*, John Wiley and Sons,1996.

[24] Z.Shi and R.B.Lee, " Bit permutation instructions for accelerating software cryptography". In *Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors*, pp. 138-148, 2000.

[25] M. Tremblay et al.,"VIS Speeds New Media Processing", In *IEEE Micro*, July/Aug. 1996, pp.10-20.

[26] Richard Weiss and Nathan Binkert "A comparison of AES candidates on the Alpha 21264". In *Proc. Third AES Candidate Conference*, April 13-14, 2000. `http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html`

[27] Thomas J. Wollinger, Min Wang, Jorge Guajardo and Christof Paar, "How Well Are High-End DSPs suited for AES Algorithms ?" In *Proc. Third AES Candidate Conference*, April 13-14, 2000. http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html

[28] H.C.A. van Tilborg,*An Introduction to Cryptology*, Boston:Kluwer Academic Publ.,1988.

[29] John Worley, Bill Worley, Tom Christian and Christopher Worley, "AES Finalists on PA-RISC and IA-64: Implementations & Performance". In *Proc. Third AES Candidate Conference*, April 13-14, 2000. http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html

[30] L.Wu, C.Weaver and T.Austin, " CryptoManiac: A Fast Flexible Architecture for Secure Communication". In *28th Annual International Symposium on Computer Architecture (ISCA-2001)*, 2001.

[31] "American National Standard for Information Systems — Data Encryption Algorithm — Modes of Operation". ANSI X3.106, American National Standards Institute, 1983.

[32] "MIPS Digital Media Extension," *Instruction Set Architecture Specification.* http://www.mips.com/publications/processor_architecture.html

[33] http://www.nist.gov/public_affairs/releases/g00-176.htm, US Commerce Department Press Release.

[34] http://www.4i2i.com/reed_solomon_software.htm

[35] http://www.4i2i.com/reed_solomon_codes.htm

[36] http://www.hifn.com

A Direct Implementation of Rijndael Encryption on Atlivec

Given below is the straight line AltiVec code for one non-final round of Rijndael encrytion.

```
 1.  vperm      V0,V11,V12,V31
 2.  vcmpgtub   V8,V31,V27
 3.  vperm      V1,V13,V14,V31
 4.  vslb       V9,V31,V28
 5.  vperm      V2,V15,V16,V31
 6.  vcmpgtub   V9,V9,V27
 7.  vperm      V3,V17,V18,V31
```

```
 8.  vslb      V10,V31,V29
 9.  vperm     V4,V19,V20,V31
 10. vcmpgtub  V10,V10,V27
 11. vperm     V5,V21,V22,V31
 12. vsel      V0,V0,V1,V10
 13. vperm     V6,V23,V24,V31
 14. vsel      V2,V2,V3,V10
 15. vperm     V7,V25,V26,V31
 16. vsel      V4,V4,V5,V10
*17. lvx       V3,G16,G4
 18. vsel      V6,V6,V7,V10
*19. lvx       V5,G17,G4
 20. vsel      V0,V0,V2,V9
*21. lvx       V10,G19,G4
 22. vsel      V4,V4,V6,V9
 23. vsel      V0,V0,V4,V8
*24. lvx       V6,G18,G4
*25. vsplitsb  V4,0
 26. vperm     V1,V0,V0,V30
*27. lvx       V7,G21,G4
*28. vcmpgtub  V8,V1,V27
*29. vslb      V9,V1,V28
*30. vsel      V0,V4,V3,V8
*31. vxor      V0,V9,V0
*32. vperm     V2,V0,V1,V5
*33. lvx       V8,G20,G4
*34. vperm     V3,V0,V1,V6
*35. vxor      V2,V2,V3
*36. vperm     V3,V0,V1,V10
*37. vxor      V2,V2,V3
 38. lvx       V5,G15,G3
*39. addi      G15,G15,16
*40. vperm     V3,V0,V1,V8
*41. vxor      V2,V2,V3
*42. vperm     V3,V0,V1,V7
*43. vxor      V2,V2,V3
 44. vxor      V31,V2,V5
```

For the purpose of the calculation of the number of cycles required for Rijndael encryption of one block, it may be noted that the marked instructions (* before the instruction number) are for *Mix Colomn* and hence would not appear in the code for the final round.

A.1 1-way AltiVec Schedule

Table 4 shows the schedule on 1-way AltiVec architecture assuming a Latency of 1 cycle for each instruction and any 3 of different types of instruction can be issued per cycle. It may be noted that in the final round instruction 38 can go in parallel with instruction 26 in cycle number 13.

A.2 2-way AltiVec Schedule

Table 5 shows the schedule on 2-way AltiVec architecture assuming a Latency of 1 cycle for each instruction and any 4 of different types of instruction can be issued per cycle. It may be noted that in the final round instruction 38 can go in parallel with instruction 26 in cycle number 8.

B Byte-Sliced Reed-Solomon Implementation on AltiVec

B.1 Encoder

The AltiVec cycles count for the primitives defined in Sec 5.2 are :

- `LDA (a, b) :` 2 cycles

- `LOAD-TABLE (a, b, c) :` 4 cycles

- `CONST-MULT (a, b, c, d) :` 3 cycles – A straightforward table lookup would require 20 instructions : this cycle count is achieved by using the method given in [14].

- `ADD (a, b, c) :` 1 cycle

The rearrangement of the data in the registers (for byte-slicing) takes 16*48 cycles. A scheduled version of the code takes [(n - k - 2) * 4 + 10] * k + 17 cycles. Thus
No. of bits output = n * 8 * 16

Table 4: Schedule of one non-final round of Rijndael encryption for 1-way AltiVec

Cycle	PERMUTE	SIMPLE	Integer	LOAD/STORE
1	1. vperm V0,V11,V12,V31	2. vcmpgtub V8,V31,V27		
2	3. vperm V1,V13,V14,V31	4. vslb V9,V31,V28		
3	5. vperm V2,V15,V16,V31	6. vcmpgtub V9,V9,V27		
4	7. vperm V3,V17,V18,V31	8. vslb V10,V31,V29		
5	9. vperm V4,V19,V20,V31	10. vcmpgtub V10,V10,V27		
6	11. vperm V5,V21,V22,V31	12. vsel V0,V0,V1,V10		
7	13. vperm V6,V23,V24,V31	14. vsel V2,V2,V3,V10		
8	15. vperm V7,V25,V26,V31	16. vsel V4,V4,V5,V10		*17. lvx V3,G16,G4
9		18. vsel V6,V6,V7,V10		*19. lvx V5,G17,G4
10		20. vsel V0,V0,V2,V9		*21. lvx V10,G19,G4
11		22. vsel V4,V4,V6,V9		*24. lvx V6,G18,G4
12		23. vsel V0,V0,V4,V8		*27. lvx V7,G21,G4
13	26. vperm V1,V0,V0,V30			
14	*25. vsplitsb V4,0	*28. vcmpgtub V8,V1,V27		
15		*29. vslb V9,V1,V28		
16		*30. vsel V0,V4,V3,V8		
17		*31. vxor V0,V9,V0		
18	*32. vperm V2,V0,V1,V5			*33. lvx V8,G20,G4
19	*34. vperm V3,V0,V1,V6			38. lvx V5,G15,G3
20	*36. vperm V3,V0,V1,V10	*35. vxor V2,V2,V3	*39. addi G15,G15,16	
21	*40. vperm V3,V0,V1,V8	*37. vxor V2,V2,V3		
22	*42. vperm V3,V0,V1,V7	*41. vxor V2,V2,V3		
23		*43. vxor V2,V2,V3		
24		44. vxor V31,V2,V5		

Table 5: Schedule of one non-final round of Rijndael encryption for 2-way AltiVec

Cycle	PERMUTE(2)	SIMPLE(2)	Integer	LOAD/STORE
1	1. vperm V0,V11,V12,V31	2. vcmpgtub V8,V31,V27		
	3. vperm V1,V13,V14,V31	4. vslb V9,V31,V28		
2	5. vpermV2,V15,V16,V31	6. vcmpgtub V9,V9,V27		
	7. vperm V3,V17,V18,V31	8. vslb V10,V31,V29		
3	9. vperm V4,V19,V20,V31	10. vcmpgtub V10,V10,V27		
	11. vperm V5,V21,V22,V31	12. vsel V0,V0,V1,V10		
4	13. vperm V6,V23,V24,V31	14. vsel V2,V2,V3,V10		
	15. vperm V7,V25,V26,V31	16. vsel V4,V4,V5,V10		
5		18. vsel V6,V6,V7,V10		*17. lvx V3,G16,G4
		20. vsel V0,V0,V2,V9		
6		22. vsel V4,V4,V6,V9		*19. lvx V5,G17,G4
7	*25. vsplitsb V4,0	23. vsel V0,V0,V4,V8		*21. lvx V10,G19,G4
8	26. vperm V1,V0,V0,V30			*24. lvx V6,G18,G4
9		*28. vcmpgtub V8,V1,V27		*27. lvx V7,G21,G4
		*29. vslb V9,V1,V28		
10		*30. vsel V0,V4,V3,V8		*33. lvx V8,G20,G4
11		*31. vxor V0,V9,V0		
12	*32. vperm V2,V0,V1,V5			
	*34. vperm V3,V0,V1,V6			
13	*36. vperm V3,V0,V1,10	*35. vxor V2,V2,V3		38. lvx V5,G15,G3
14	*40. vperm V3,V0,V1,V8	*37. vxor V2,V2,V3	*39. addi G15,G15,16	
15	*42. vperm V3,V0,V1,V7	*41. vxor V2,V2,V3		
16		*43. vxor V2,V2,V3		
17		46. vxor V31,V2,V5		

No of cycles = [(n - k - 2)* 4 + 10] * k + 17
Throughput = (n * 8 * 16)/{ [(n - k - 2) * 4 + 10] * k + 17} bits per cycle
= (n * 8 * 16)/{ [(n - k - 2) * 4 + 10] * k + 17}* (clock speed) bits per sec

For n = 255 k = 239 ,the scheduled version of the same takes about 16000 cycles, thus yielding a throughput of around 2.06 * clock speed = 273 Mbps on 133 Mhz clock machine.

B.2 decoder

This is how the three steps of decoding , Sec 2.1, are implemented (the numbers are for RS(255,239), unless mentione otherwise) –

- Syndrome Calculation : We give the code for calculation of syndromes in terms of the primitives as used in the encoder.

  The following sybmols are used in the pseudo-code : R1, R2, R3, R4, S are the vector registers used and r_i is the i_{th} received symbol.
  Pseudo - Code:

  1 : `LOAD-TABLE( R2,R3,R4)`
  2 : `LDA` $(r_{n-1}, R1)$
  3 : `CONST-MULT (P,R1,R3,R4)`
  4 : `LDA` $(r_{n-2}, R1)$
  5 : `ADD (S,P,R1)`
  6 : `CONST-MULT (P,S,R3,R4)`

  After this 4,5 and 6 step would be repeated 252 times i.e. total number of step for one $S_i = (2+1+3) * 253$

  7 : `LDA` $(r_{n-3}, R1)$
  8 : `ADD (S,P,R1)`
  ...
  ...
  763 : `LDA` $(r_0, R1)$
  764 : `ADD (S,P,R1)`

  Total number of cycles for computing 1 S_i=4+1+3+253(2+1+3)+2+1=1529
  Total number of cycle for 16 S_i = 24,464 [(n - 1) * (n - k) * 6 in the general case]

- Calculation of the error- location polynomial : We give the code for calculation of syndromes in terms of the following primitives:
  - Update-Sigma() : Updates the value of the error location polynomial, σ_μ at each iteration.
  - Update-Lu() : updates l_μ (the degree of error location polynomial) and ρ_μ if necessary.
  - Update-Du() : computes the discrepancy d_μ at each stage

Pseudo Code :

```
for (i=1;i<=16;++i) {
    Update-Sigma();
    Update-Lu();
    Update-Du();
}
```

The primitves are described in more detail below.

```
Update-Sigma() {
      t1=inverse(dμ) ;
      t2=mult-form(dμ) ;
      mult(t1,t2);
      sub(μ − ρ) ;
      for (i=1; i≤8 ; ++i) {
         (check which of the eight registers the current entry has to go)
      }
}
```

Update-sigma() takes 50 + 50 + 10 + 2 + 8* (8*6 + 4) = 528 cycles

```
Update-Lu() {
      l_{μ+1} = max(l_μ, lρ + μ − ρ)
       update(r);
}
```

Update-Lu() takes 14 + 8*3 + 5 = 43 cycles

```
Update-Du() {
      for(i=1; i≤ 8;++ i) {
```

```
        t1=mult-form(Si);
        t2=mult-form(sl);
        t3=mult(t1,t2);
        t4=add-form(t3);
        add to sum
    }
}
```

Update-Du() takes 3+ 8* 2 + 50 +50 +10 +2 + 50 + 3 +3 = 1342 cycles.
Thus, **total no of cycles for this step** = 16 * 528 + 43 +1342 = 30608
[≈ {(n - k) * (n - k)/2 [3*(n - k) + 174] } +137 in the general case]

- Finding Error location numbers and error values :

Step 1 : To find the roots of the error-location polynomial :

In 8 registers we store the value of $\sigma_1, \sigma_2, ..., \sigma_8$
In 16 registers we store constant multiplication table for $\alpha_1, \alpha_2, ..., \alpha_8$.

At each iteration we have : 8 CONST-MULT(24 cycles),8 ADD(8 cycles),1 COMPARE (1 cycle),1 NOR(1 cycle), 1 AND(1 cycle) and 1 STORE (2 cycles).

Thus, total number of cycles required = 254(24+8+1+1+1+2)=9,398
[≈ n * { 4 * (n - k)/2 + 5} in the general case].

Step 2 : Computation of coefficients of Z(X)

Here we have 8 Load σ_i(16 Cycles), 8 load with S_i(16 Cycles) and 8 registers are assigned values : $S_1 + \sigma_1, S_2 + \sigma_1 * S_1 + \sigma_2, ...$
Total Cycles required = 1+(2+60)+(3+2.60)+......+(9+8.60)= 2205 [≈ (n-k)/2 +1 *{ 31 (n- k)/2 +2} in the general case].

Step3 : Calculation of Error value at location $\beta_l = \alpha^{jl}$.

There are 3 steps here –

– Inverse calculation of β_l (50 cycles)

- Evaluation of Z(X) $= 1 + (S_1 + \sigma_1)X + (S_2 + \sigma_1 * S_1 + \sigma_2) * X^2 + ... + (S_v + \sigma_1 * S_{v-1} + \sigma_2 * S_{v-2} + ... + \sigma_v)X^v = 1 + Z_1X + Z_2X^2 + ... + Z_vX^v$ at X $= \beta_1, \beta_2, ..., \beta_8$

  (Note: $Z_i's$ have been computed in the previous step)
  Calculate mult-form of β_i(50 cycles)
  The next set of instructions is carried out 8 times for each β_i.
    Calculate the mult-form of Z_8 (50 cycles)
    Multiply it with β_i (10 cycles)
    Calculate the results' add form (50 cycles)
    Add it to Z_7 (1 cycle)

  Total number of cycles=8*(50+10+50+1)=888.Storing of the error location numbers would require another (3*8 +1)* 255 = 6375 number of cycles.

- There are are 3 steps here –
  Calculation of $\prod(1 + \beta_i\beta_l^{-1})$ (51*8+50*7=758 cycles).
  Inverse Calculation of $[\prod(1 + \beta_i\beta_l^{-1})]$ (50 cycles).
  Multiplication by $[\prod(1 + \beta_i\beta_l^{-1})^{-1}]$ (50 cycles).

Total number of cycles required for e_{jl} is 888+758+50+50=1746 cycles. Thus, eight $e'_{jl}s$ requires 13,968 cycles
Thus,a total of 9,398 + 32 + 2205 + 13,968 + 6375 = 31,978 cyles are required for the last step of the decoder.

Thus the entire decoder takes (24464 +30608 +31978) = 87050 cycles.
Number of output bits = 239 * 8 *16 = 30592
Thus throughput = (30592/87050) * clock speed
For clock speed = 133 MHz this turns out to be around 46 Mbps.