Efficient Galois Field Arithmetic on SIMD Architectures*

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

Abstract

We propose techniques to utilize the data parallelism capabilities of a SIMD architecture in computations involving Galois Field arithmetic. Galois Field arithmetic finds wide use in engineering applications, including error-correcting codes and cryptography. Often these applications involve extensive arithmetic on small (8-bit) numbers, and straightforward implementations may highly under-utilize the wide-word capabilities of a SIMD processor.

So far, we have only seen limited, application-specific implementations of Galois Field based applications targeting SIMD architectures. In this paper we present techniques and an algorithm which, given the specification of the computation, produce an efficient SIMD implementation. To evaluate the quality of the output of our algorithm, we applied it to two applications which use Galois Field arithmetic: Rijndael block cipher [12] and Reed-Solomon error correcting codes [20]. We got significant performance improvements for both the applications. Our algorithm uses bit-slicing [8] and it's generalization data-slicing to fully explore the full potential of SIMD instructions. The implementations require minimal architectural support from any wide-path SIMD processor: parallel table lookup, bitwise EXOR/AND and LOAD/STORE operations. Further, in the case of bit-slicing only the last four instructions are required. This has the advantage of portability of our implementations across wider range of SIMD architectures.

1 Introduction

SIMD architectures, such as the AltiVec extension to PowerPC[13], 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 [28] and SSE2[14], Sun's VIS [34] and AMD's 3DNow! [25].

Galois Field Arithmetic finds wide use in engineering applications such as error-control codes [9, 22], cryptography [35], switching theory [7] and digital signal processing [24, 29]. Computationally, Galois Fields are attractive in several ways. All operations are carry-less, the word length is constant and there are no rounding issues.

*This work was done when the authors were at IBM India Research Lab, New Delhi, India.

†Department of Mathematics, Indian Institute of technology, New Delhi, India.
‡Broadcom Corporation, CA, Email: `pkdubey@broadcom.com`
§Strategic Planning and Optimization Team, Amazon.com, Seattle, WA, Email: `vijayk@amazon.com`
¶Department of Computer Science, University of Texas at Austin, Austin, TX. Email: `atri@cs.utexas.edu`
||Department of Mathematics, Indian Institute of Technology, New Delhi, India.

Obtaining efficient implementations on SIMD architectures has so far been an application-specific task. In this paper we show that SIMD architectures are a natural fit for Galois Field arithmetic, and propose a methodology to fully exploit the parallelism potential of SIMD architectures. Several aspects of Galois Field arithmetic make it suitable for SIMD implementations. Galois Fields of the form $GF(2^k)$ arithmetic involve 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. In addition, k is usually a multiple of 8 and often a power of 2, which is convenient from the point of view of subword parallelism. Further, we can define special Galois Fields (called *composite fields* — see Section 3.1) wherein we can meaningfully consider the k-bit input as m contiguous n-bit groups (where $k = nm$) which nicely corresponds to instructions in SIMD architectures like AltiVec [13] which can can logically divide a wide data word into packed subwords.

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 an algorithm to transform Galois Field arithmetic-based computations into efficient SIMD implementations. To our knowledge, we are not aware of any previous work to automatically generate SIMD implementations for computations involving Galois Field arithmetic. To evaluate the performance of our algorithm, we select two representative and perhaps two of the most popular applications of Galois Field arithmetic: Reed-Solomon error correcting codes and the Rijndael Block Cipher [12]. Rijndael cipher has been selected by the US National Institute of Standards and Technology(NIST) as the new Advanced Encryption Standard(AES). 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 [8] 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 organized as follows. In Section 2, we describe Galois Field Arithmetic, Rijndael block cipher and Reed-Solomon error correcting codes. Section 3 talks about techniques which we use in our algorithm (Section 4). Finally in Section 5 we describe the Rijndael and Reed-Solomon implementations obtained from the algorithm. We conclude in Section 6, and Section 7 contains figures and illustrations.

2 Galois Field arithmetic

A Galois Field of q elements (denoted by $GF(q)$) is a set of q elements (generally represented by the set $\{0, 1, 2, \cdots, q-1\}$) in which addition, subtraction, multiplication and division (all appropriately defined) can be performed without leaving the set. Addition and multiplication must satisfy the commutative, associative and distributive laws. Galois Fields with $q = 2^k$ for some integer $k > 0$ are more popular as the elements can be stored using k bits. Addition in these fields is just the bit-wise EXOR of the k-bit operands. Multiplication can be performed in three steps — the operands (which are in their *additive-form*) to their *multiplicative-form*, addition modulo $(q-1)$ is performed on the converted operands and finally the result is converted to its additive-form to obtain the product. Figure 1 illustrates the above for $GF(2^2)$.

The k bit representation is often regarded in Galois Field literature as a $k-1$ degree polynomial with coefficients from $GF(2)$ (that is, either 0 or 1). Addition in $GF(2^k)$ is then a simple polynomial addition (that is, EXORing the corresponding coefficients) while $GF(2^k)$ multiplication is polynomial multiplication modulo some "prime" polynomial of degree k (which again takes coefficients from $GF(2)$). $GF(2^k)$ is said to be "constructed" from $GF(2)$ by this "prime" polynomial. When $q = 2^k$, the (boolean) circuit for any operation consists of just AND and EXOR gates, which makes Galois Fields attractive from the point of view of *bit-slicing* (Section 3.2). Readers who are interested in a comprehensive exposition of Galois Fields are referred to [19].

In the next two sub-sections we give an overview of Reed-Solomon Error Correcting Codes [20] and Rijndael Block Cipher [12], the two representative Galois Field applications that are the focus of this paper.

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. The Reed-Solomon decoder processes each block and attempts to correct errors that occur in transmission or storage. The number and type of errors that can be corrected depends on the characteristics of the Reed-Solomon code used.

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. This code is capable of correcting any combination of t or fewer errors in a block of n symbols, where $(n - k) = 2t$, 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 elements 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 $g(x)$ 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).

Berlekamp's algorithm is used to find the error-location polynomial. See [1, 20] for more details.

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

2.2 The 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. Each plain text block (as also the intermediate cipher block) consists 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 as shown in Figure 2, where there are 10 *rounds* each of which is as shown in Figure 3.

See [12] for the detailed specifications of Rijndael.

3 Techniques for parallelism

3.1 Composite fields

Composite fields are frequently used in implementations of Galois Field arithmetic [23, 26, 27]. In cases where arithmetic operations rely on table lookups, composite field 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 objective has usually been to obtain better software implementations by using smaller tables through subfield arithmetic. Applications to hardware design (such as [27]) have been relatively infrequent.

The motivation behind the use of composite fields is to somehow divide the k-bit operands (from $GF(2^k)$) into m contiguous n-bit blocks (where $k = nm$) and operate upon the new n-bit operands (which are elements of $GF(2^n)$) with the hope that the total cost of the computation would decrease (of course there is a trade-off between n and m). As an analogy, consider 8-bit addition. One can use an 8-bit adder to do the addition or use two 4-bit adders or four 2-bit adders or eight 1-bit adder (in the last three cases one would have to take care of the carries and overflows). Now the best way to implement the addition depends upon how efficient the adders are, the number of additions to be performed and how much the overhead of taking care of the overflows and carries amounts to. The situation in the composite fields is similar except one needs to do a transformation on the operands to get a meaningful partition of the original input bits. Thus, the overhead in this case is this transformation cost (recall that unlike integer arithmetic, Galois Field arithmetic does not have overflow or carry issues).

Formally, after [26], 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, as discussed above. For a given underlying field $GF(2^k)$, our gains depend on the choice of n, m and the polynomials $Q(y)$ and $P(x)$.

3.2 Slicing techniques

Bit-slicing was introduced by Biham [8] to obtain a fast implementation of DES [10] in software. 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. If this simulation requires N instructions, then each computation costs $\frac{N}{W}$ instructions. As mentioned in Section 2, computations in Galois Field arithmetic have the property that all the gates are AND and EXOR, which translates to single-cycle instructions (unlike general arithmetic where one may have to use more expensive instructions).

We will illustrate the technique of bit-slicing with a simple example of multiplication in the field $GF(2^2)$. Figure 4 gives the table specifying the multiplication[2] operation in $GF(2^2)$.

Figure 5 shows how multiplication can be implemented by converting the inputs[3], which are in the additive form, to their multiplicative form, adding them modulo 3 and then converting the result back to the additive form.

It can seen easily that the boolean equations[4] $c_1 = (b_1 \cdot (a_0 \oplus a_1)) \oplus (a_1 \cdot b_0)$ and $c_0 = (a_1 \cdot b_1) \oplus (a_0 \cdot b_0)$ satisfy the multiplication table in Figure 4. For our example we will assume word width $W = 2$. So we "simulate" two multiplication instances ($C = A\mathrm{X}B$ and $c = a\mathrm{X}b$) as shown in Figure 6.

To get an idea of the possible speed-up[5] obtained when using bit slicing, consider multiplication in $GF(2^8)$ on AltiVec [13]. 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 may require at least 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!

One of the advantages of SIMD architectures like AltiVec [13] is that they offer different "granularity" of operations (for example in AltiVec, the 128-bit word can treated as two 64-bit operands, four 32-bit operands, eight 16-bit operands or sixteen 8-bit operands). The table lookup technique in Figure 5 corresponds to the "coarsest" granularity while bit-slicing corresponds to the "finest" granularity. Thus, the intermediate levels of granularity are not investigated in these implementations. We propose the use of (d-bit) *data-slicing* to explore the full potential of SIMD instruction set architectures. Here instead of grouping the corresponding bits together (as we did in bit-slicing) we will divide the inputs into partitions of (contiguous) d-bits and group the corresponding partitions together. Further all the instructions treat the W-bit word as $\frac{W}{d}$ operands of d bits each (instead of W single bit operands in bit-slicing). In the next sub-section we see what factors need to be taken into consideration while choosing the right slice width and how the different levels of granularity

[2]The four elements in $GF(2^2)$ can be represented using 2-bit integers, and thus they correspond to the set $\{0, 1, 2, 3\}$.

[3]Subscript 0 denotes the least significant bit.

[4]$\oplus$ denotes the EXOR operation and $\cdot$ denotes the AND operation.

[5]The figures are from the Rijndael implementation in [30].

correspond to composite fields of Section 3.1.

3.3 Combining the two techniques

For k-bit inputs such that $W > k$, the first level of granularity which makes sense to consider is $d = k$. For finer levels of granularity, for example $d = \frac{k}{m}$, we should be able to meaningfully divide the k-bit input into m $\frac{k}{m}$-bit operands. As discussed in Section 3.1, this corresponds to converting elements from $GF(2^k)$ to $GF((2^n)^m)$, where $k = nm$.

Let us first see an intuitive explanation of why data-slicing can give good performance. Recall the example of $GF(2^2)$ multiplication in the last section. In each step, a 2-bit operand undergoes some transformation. However, not *all* bits are involved in each step (of Figure 5). So if we have 2 $GF(2^2)$ multiplications in parallel, the same bits in both the instances would be unaffected. On the other hand, in bit slicing only those bits (from both the instances) are operated upon which are involved in the operation- this translates into fewer instructions for the bit-sliced implementation.

However, the performance 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 which would defeat the purpose of slicing. The placement of the unused bits in the inputs depends on the underlying computation and can vary from one to another.

The other important factor determining the performance is the architectural support at different levels of granularity. Fixing the slice width at d makes sense only if the required instruction can view the W-bit word as $\frac{W}{d}$ d-bit operands. Further, instructions for the same operation at different level of granularity can have different execution times. Another overhead which must be taken into account while using composite fields is the cost of transforming the elements in the original field to the composite field and back.

The proposed methodology first determines the possible slice sizes (that is value of d, which in turns fixes the values $n = d$ and $m = \frac{k}{d}$). For each value of d, the composite field $GF((2^d)^{\frac{k}{d}})$ is determined by the possible choices of the polynomials $Q(y)$ and $P(x)$. Thus, choosing d, $Q(y)$ and $P(x)$ fixes all the parameters in an implementation. We also fix a cost function to evaluate each implementation (for example the number of instructions or an estimate of the number of cycles required) and then we choose the implementation which minimizes the cost (as determined by the cost function) across all possible values of the triple $(d, Q(y), P(x))$.

As mentioned in Section 1, the data-sliced implementation has good portability properties: in general, the only instructions required are parallel table lookup, bitwise EXOR and AND, LOAD and STORE operations. Further, in the case of bit-slicing, only the last four instructions are required.

4 Automating the process

In this section we briefly sketch how the method in Section 3.3 for obtaining an efficient implementation can be automated– that is, given a specification of the computation in terms of addition, multiplication and inversion operations in some $GF(2^k)$ [6], the algorithm given below would output a sequence of instructions to efficiently implement the computation. The algorithm builds upon the ideas presented in Section 3.

Let $\mathcal{S}$ denote the sequence of $GF(2^k)$ operations which specify the computation. get_irr_polys_simple(d) returns the set of degree-d irreducible polynomials over $GF(2)$, get_irr_polys_cmplx($d, n, P(x)$) returns the set of degree-d irreducible polynomials over $GF(2^n)$ (generated by $P(x)$) and compile($d, P(x), Q(y), \mathcal{S}$) returns a sequence of instructions from the underlying SIMD architecture (of datawidth W-bits) which implements d-bit data-slicing in $GF((2^d)^{\frac{k}{d}})$ (where the underlying polynomials are $P(x)$ and $Q(y)$). Given the inputs k and $\mathcal{S}$, the algorithm proceeds as follows to output the sequence of instructions $\mathcal{I}$:

1. Set $d \leftarrow k$, $\mathcal{I} \leftarrow \emptyset$, $min_c \leftarrow \infty$.
2. If $k > W$ exit.
3. Repeat forever Steps 4 to 7:
4. While (d does not divide k) set $d \leftarrow d - 1$.
5. If $d = 0$ exit.
6. $\mathcal{P} \leftarrow$ get_irr_polys_simple($d - 1$).
7. For all polynomials $P(x)$ in $\mathcal{P}$ do
   - $\mathcal{Q} \leftarrow$ get_irr_polys_cmplx ($\frac{k}{d} - 1, n, P(x)$).
   - For all polynomials $Q(y)$ in $\mathcal{Q}$ do
     - $I_{P(x),Q(y),d} \leftarrow$ compile($d, P(x), Q(y), \mathcal{S}$).
     - if ($min_c >$ cost($I_{P(x),Q(y),d}$)) then
       $min_c \leftarrow$ cost($I_{P(x),Q(y),d}$) and $\mathcal{I} \leftarrow I_{P(x),Q(y),d}$

We complete the algorithm by sketching how to implement the procedures for even values of k and slice sizes k, $\frac{k}{2}$ and 1:

- get_irr_polys_simple(d): The set of irreducible polynomials of degree d over $GF(2)$ can be obtained from Maple [4] or using tables [31].
- get_irr_polys_cmplx($d, n, P(x)$): We are considering only the cases $d = k - 1$ (in which case return the empty set), $d = \frac{k}{2} - 1$ (from [26, 19], return the set of polynomials of the form $Q(y) = y^2 + y + \lambda$, $\lambda \in GF(2^n)$ such that $Q(y)$ has no roots in $GF(2^n)$) and $d = 1$ (Using techniques from [26] we can output a boolean circuit using only EXOR and AND gates).
- compile($d, P(x), Q(y), \mathcal{S}$): As mentioned in Section 3.3, the only instructions required are parallel table lookups, bit-wise EXOR/AND and LOAD/STORE to implement all the operations.
- cost(I): This can be simply the number of instructions in I or an estimate of the number of clock cycles required by the instructions in I.

In the next section we present the performance results of our Rijndael and Reed-Solomon implementations.

[6] Subtraction is the same as addition in $GF(2^k)$.

5 Data-sliced implementation

5.1 Bit-sliced Rijndael Implementation

We obtained significant speed-up for Rijndael encryption using the bit-slicing. As mentioned in Section 3.2, this implied designing an efficient hardware circuit for the same. The details of this implementation have been reported in [30].

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 [5]) to maintain 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. [37] discussed two such scenarios: network traffic experiments and secure disk experiments. However, a new invention in [17] gives a parallel mode of encryption while (provably) maintaining the same security level.

5.1.1 Performance and Analysis

Table 5.1.1 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[18] 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 [32, 11]. For software-based VPN/firewall or SSL servers, this can potentially save the cost of any external security coprocessor or crypto accelerator [37, 3]: an external security engine often has a protocol communication overhead with the host server processor.

5.2 Byte-sliced Reed-Solomon Implementation

We have implemented Reed–Solomon Codes for $m = 8$. Details are deferred to Appendix B.

Table 2 compares the throughput of Reed–Solomon (encoding and decoding) implementations. The performance number of Henrion [15] are for a 133MHz Pentium while that of 4i2i.com [2] is for a 166 MHz Pentium. Our implementation is for a comparable PowerPC processor implementation

Table 1: Cycle counts per block for Rijndael encryption implementations

	Cycles	Architecture	Comments
Worley et al[16]	284	Pentium	Requires an 8KB table
	176	PA-RISC	
	124	IA-64	
Weiss et al.[36]	210	Alpha 21264	
Wollinger et al.[33]	228	TMS320C6x	
Aoki et al.[6]	237	Pentium II	
Cryptomaniac [37]	90	external co-processor	
AltiVec (Appendix A)	230	1-way Altivec	Requires only a 256 byte table
	162	2-way Altivec	
Bit-sliced implementation	170	256b datapath width	Requires only EXOR, AND, L/S and 2KB table
	119	384b datapath width	
	100	512b datapath width	

Table 2: Throughput of Reed-Solomon Implementations in Mbps

	RS(255,251)		RS(255,239)	
	Encoding	Decoding	Encoding	Decoding
Henrion[15] (133 MHz Pentium)	45		NA	
4i2i.com[2] (166 MHz Pentium)	12 (Encoding & Decoding)		2.72 (Encoding & Decoding)	
Byte-Sliced AltiVec Implementation (133 MHz PowerPC)	956	332	273	46

of the AltiVec architecture at 133 MHz. It maybe noted that there were no separate figures for encoding and decoding for [2]. To make a meaningful comparison with our results, note that if there are separate encoding block (which takes t_1 time to code one data block) and decoding blocks (which takes t_2 time to decide one data block), then when the blocks operate in serial, the total time taken is at most $(t_1 + t_2)$. Thus, out figures for Encoding & Decoding would be 246.4 and 39.4 cycles for RS(255,251) and RS(255,239) respectively. As shown in Table 3, byte-sliced implementation offers more than 14 fold speedup over previously published numbers for general-purpose processors.

6 Conclusions

We have presented an algorithm which outputs an efficient implementation of applications using Galois Field arithmetic on SIMD architectures. This algorithm combines the techniques of data-slicing and composite fields to fully exploit the potential of SIMD instructions. An interesting feature of the implementation returned by the algorithm 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. Finally, using the implementations given by our algorithm, we achieved significant improvement in the performance of Rijndael encryption and Reed-Solomon codecs.

7 Illustrations

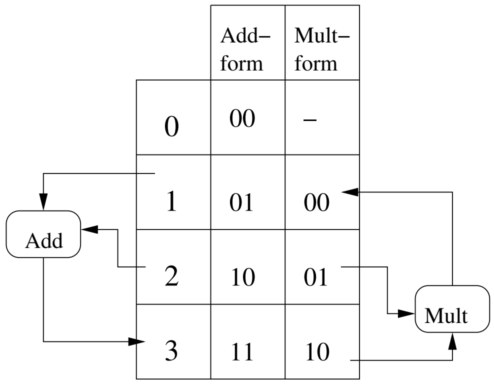

Figure 1: The Galois Field $GF(2^2)$. Both the *additive-form* and the *multiplicative-form* of the elements are shown. Addition and multiplication are illustrated with an example each.

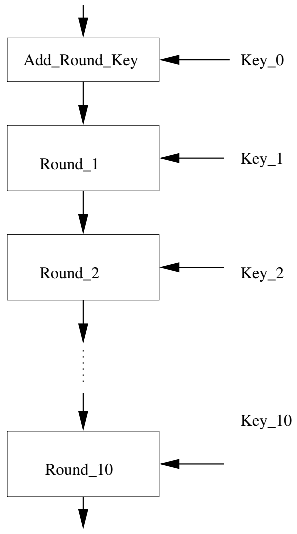

Figure 2: A high-level view of Rijndael [12]. Each data path is 128 bits wide. *Add_Round_Key* is the bit-wise EXOR of the two inputs.

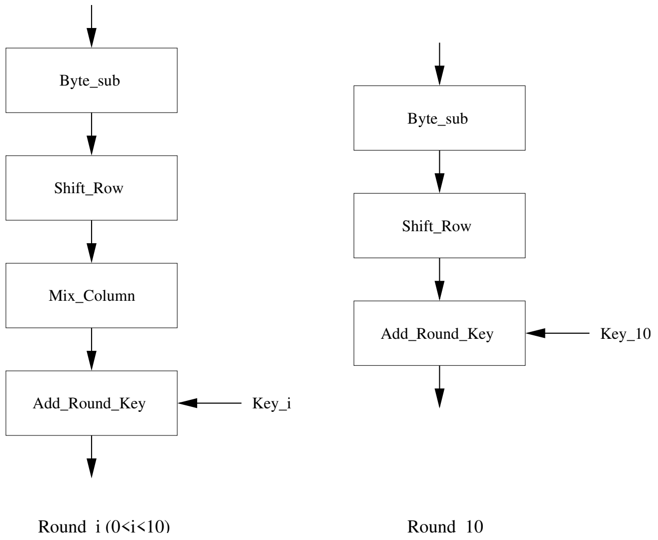

Figure 3: $Round_i$, $1 \leq i \leq 10$ of Figure 2. Each data path is 128 bits wide.

X	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

Figure 4: Multiplication in $GF(2^2)$

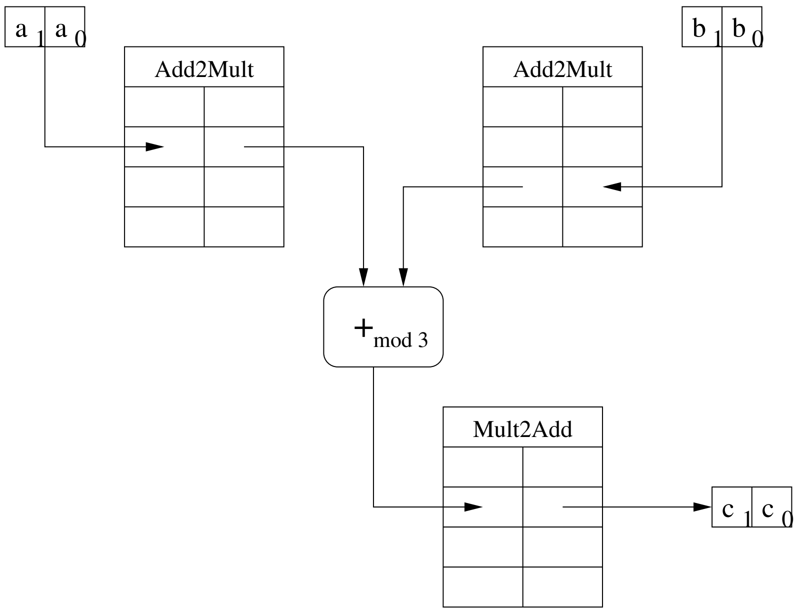

Figure 5: Multiplication in $GF(2^2)$ ($c = a\mathrm{X}b$) using Table lookups. *Add2Mult* is the table to convert elements of $GF(2^2)$ from their additive-form to their multiplicative form. *Mult2Add* does the inverse.

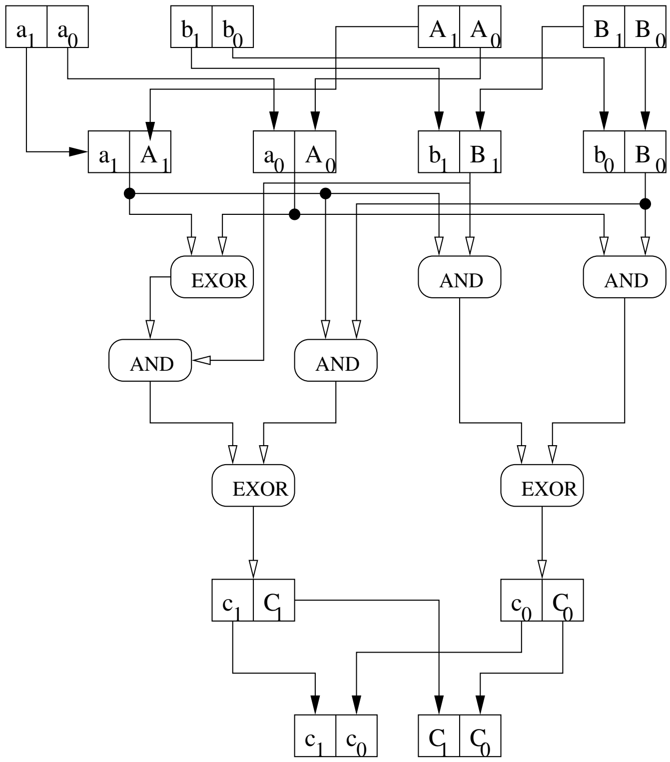

Figure 6: Multiplication in $GF(2^2)$ ($c = a\mathrm{X}b$ and $C = A\mathrm{X}B$) using *bit-slicing*. *EXOR* takes two 2-bit inputs and outputs their bit-wise EXOR. *AND* computes the bit-wise AND. The arrows with a hollow head denote a width of 2 bits while solid arrow-heads denote a width of 1 bit.

References

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

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

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

[4] http://www.maplesoft.com/main.shtml .

[5] American national standard for information systems — data encryption algorithm — modes of operation. In *ANSI X3.106,American National Standards Institute*, 1983.

[6] K. Aoki and H. Lipmaa. Fast implementations of aes candidates. In *Proc. Third AES Candidate Conference*, April 13-14, 2000.

[7] B. Benjauthrit and I. S. Reed. Galois switching functions and their applications. *IEEE Trans. Comput.*, C-25:78–86, January 1976.

[8] E. Biham. A fast new des implementation in software. In *Proceedings of Fast Software Encryption 4*, 1997.

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

[10] N. Bureau. *Data Encryption Standard.* U.S. Department of Commerce, FIPS 46, 1977.

[11] 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.

[12] J. Daemen and V. Rijmen. Aes proposal: Rijndael. In http://www.esat.kuleuven.ac.be/~rijmen/rijndael/index.html, 2000.

[13] K. Diefendroff, P. K. Dubey, R. Hochsprung, and H. Scales. Altivec extension to powerpc accelerates media processing. *IEEE Micro*, pages 85–95, March-April 2000.

[14] P. Glaskowsky. Pentium 4 (partially) preview. In *Microprocessor report*, 2000.

[15] J. C. Henrion. An efficient software implementation of a fec code. In *Proc. of IDMS'97*, September 1997.

[16] T. C. John Worley, Bill Worley and C. Worley. Aes finalists on pa-risc and ia-64: Implementations & performance. In *Proc. Third AES Candidate Conference*, April 13-14, 2000.

[17] C. S. Jutla. Encryption modes with almost free message integrity. In *Proc. of Eurocrypt*, 2001.

[18] K. Krewell. Amd takes hammer to itanium. *Microprocessor Report*, Nov. 26, 2001.

[19] R. Lidl and H. Niederreiter. *Introduction to finite fields and their applications.* Cambridge University Press, Cambridge, Ma., 1986.

[20] S. Lin and D. J. Castello. *Error Control Coding: Fundamentals and Applications.* Prentice Hall, 1983.

[21] B. Lucier. Cryptography, finite fields, and altivec. In `http://www.simdtech.org/apps/group_public/download.php/22/Cryptography.pdf`.

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

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

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

[25] S. Oberman, G. Favor, and F. Weber. Amd 3dnow! technology: Architecture and implementations. *IEEE Micro*, 19:37–48, March-April 2000.

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

[27] C. Paar and P. Soria-Rodriguez. Fast arithmetic architectures for public-key algorithms over galois fields $gf((2^n)^m)$. In *Proceeding of Eurocrypt*, 1997.

[28] A. Peleg and U. Weiser. Mmx technology extension to the intel architecture. *IEEE Micro*, pages 42–50, July/Aug. 1996.

[29] I. S. Reed and T. K. Truong. The use of finite fields to compute convolutions. *IEEE Trans. Inform. Theory*, IT-21,No.2:208–213, March 1975.

[30] A. Rudra, P. K. Dubey, C. Jutla, V. Kumar, J. R. Rao, and P. Rohatgi. Efficient implementation of rijndael encryption with composite field arithmetic. In *Proceedings of the Third Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2001.

[31] G. Seroussi. Table of low-weight binary irreducible polynomials. Technical Report HPL-98-135, Computer Systems Laboratory, HP, 1998.

[32] 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*, 2000.

[33] J. G. Thomas J. Wollinger, Min Wang and C. Paar. How well are high-end dsps suited for aes algorithms ? In *Proc. Third AES Candidate Conference*, April 13-14, 2000.

[34] M. Tremblay, J. M. O'Connor, V. Narayan, and L. He. Vis speeds new media processing. *IEEE Micro*, pages 10–20, July/Aug. 1996.

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

[36] R. Weiss and N. Binkert. A comparison of aes candidates on the alpha 21264. In *Proc. Third AES Candidate Conference*, April 13-14, 2000.

[37] 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.

A Direct Implementation of Rijndael Encryption on AltiVec

We selected AltiVec [13] as our SIMD architecture to implement the Rijndael algorithm. AltiVec has a bank of thirty two 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[7] lookup using the *permute* instruction. In the specification of Rijndael, each *state* is looked upon as a 4×4 matrix. The state is assumed to be stored in column major form in a vector register.

In this section, a *state* will be denoted as S and each byte will be referred to as S_{ij} with reference to the matrix representation of the *state*. Any column in the matrix representation of a *state* will be referred 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 column transformation and the latter for the output.

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*.

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

```
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
```

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

```
  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.

Table 3: 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		

A.1 1-way AltiVec Schedule

Table 3 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 4 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.

Table 4: 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		

B Byte-Sliced Reed-Solomon Implementation on AltiVec

B.1 Encoder

Typically, a Reed–Solomon Encoder is implemented in a LFSR-fashion [20]. In our byte-sliced version we pack the vector register with the corresponding input bytes of 16 independent streams and simulate 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[8] 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 $(n-k)$: $(n-k) \leq 28$[9] i.e. at most 14 errors can be corrected with this design.

The code:

```
LDA (i_{k-1}, I_o)
ADD (R1, I_o, S_{n-k-1})
LOAD-TABLE (g_{n-k-1}, R_{g1}, R_{g2})
CONST-MULT (S_{n-k-1}, R1, R_{g1}, R_{g2})
ADD (S_{n-k-1}, S_{n-k-2}, S_{n-k-1})
LOAD-TABLE (g_{n-k-2}, R_{g1}, R_{g2})
CONST-MULT (S_{n-k-2}, R1, R_{g1}, R_{g2})
ADD (S_{n-k-2}, S_{n-k-3}, S_{n-k-2})
...
...
CONST-MULT (S_0, R1, R_{g1}, R_{g2})
```

The above code is repeated k times.

The AltiVec cycles count for the primitives defined above are :

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

[8] The table has 32 byte entries

[9] 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.

- 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 [21].

- 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
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 mentioned otherwise) –

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

  The following symbols 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 primitives 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: $Z_i^{\prime}s$ 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 cycles 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.