Vectorization of Reed Solomon Decoding and Mapping on the EVP

Akash Kumar[1], Kees van Berkel[1,2]
[1]Eindhoven University of Technology, Eindhoven, The Netherlands
[2]NXP Semiconductors, Eindhoven, The Netherlands
a.kumar@tue.nl

Abstract

Reed Solomon (RS) codes are used in a variety of (wireless) communication systems. Although commonly implemented in dedicated hardware, this paper explores the mapping of high-throughput RS decoding on vector DSPs. The four modules of such a decoder, viz. Syndrome Computation, Key Equation Solver, Chien Search, and Forney pose different vectorization challenges. Their vectorizations are explained in detail, including optimizations specific for Embedded Vector Processor (EVP). For RS (255,239), this solution is benchmarked vs published implementations, and scalability up to vector size 64 is explored. The best and the worst case throughput of our implementation is 8 times and 2 times higher respectively than other architectures.

1. Introduction

Wireless radio standards (for cellular, broadcast, connectivity, and positioning) are proliferating rapidly and evolving continuously. Accordingly, the trend in mobile handsets is toward multi-standard architectures, that support a range of radio standards through SW configurability and programmability, or a combination of the two.

Most digital wireless standards require some form of error correction, based, for example, on Reed Solomon (RS) codes. Allocating fixed hardware resources for a RS decoder would be wasteful for standards not using this decoder. In such case it would be more efficient to map the decoder function on a more generic, programmable DSP. However, with a few dozen of operations per output sample, and sample rates of 10-100 MHz, RS decoding is rather computational intensive.

Wide SIMD machines, also known as vector processors, offer the required compute power. However, the available parallelism is constrained to SIMD operations. So-called *vectorization*, in computer science, is the process of converting a computer program from a scalar implementation, which does an operation on a pair of operands at a time, to a vectorized program where a single instruction can perform multiple operations on a pair of vector (series of values adjacent in memory) operands.

In this paper, we discuss the vectorization of RS decoding, with special attention to scalability in vector sizes up to 64. RS codes and the various modules of RS decoders are introduced in Section 2. It turns out that these different modules require different approaches to vectorization (Section 4). The EVP is introduced (Section 3) as an advanced example of a vector processor, also suggesting some EVP-specific optimizations. In the result Section (Section 5) our results are benchmarked versus other implementations.

2. Reed Solomon Codes

Reed Solomon codes are a subset of Bose-Chaudhuri-Hocquenghem (*BCH*) codes and are linear block codes [1]. A Reed Solomon code is specified as $RS(n, k)$ with s-bit symbols. This implies that for every k input symbols, $n-k$ parity symbols are added to make an n-symbol code word. Such a code word is able to correct errors in up to t symbols, where $2t = n - k$. An $RS(255, 239)$ code adds 16 parity symbols for every 239 input symbols to make 255 symbols overall of 8 bits each, and is able to correct errors in 8 symbols. Figure 1 shows an example of a systematic RS code word. In a systematic code-word, input symbols are left unchanged and parity symbols are appended to it.

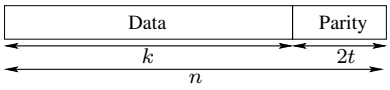

Figure 1. A typical RS code word

Noise in the transmission channel often introduces errors in the code word. Reed-Solomon codes are particularly well suited to correcting burst errors i.e. where a series of bits in the codeword are received in error. Let us say if $r(x)$ is the received code word, we get

$$r(x) = c(x) + e(x) \quad (1)$$

where $c(x)$ is the original code word and $e(x)$ is the error introduced in the system. The aim of the decoder is to find $e(x)$ and then subtract it from $r(x)$ to recover original code word transmitted. It should be added that there are two aspects of decoding - error detection and error correction. In RS code, the error can only be corrected if there are at most t errors. More than t errors may be detected, but it is not possible to correct them.

2.1. Decoder Structure

A detailed explanation on Reed Solomon decoders can be found in [1] and [2]. The decoder consists of four modules as shown in Figure 2. The first module computes the *syndrome polynomial* from the code word. The syndromes are used to construct an equation which is solved in the next

978-3-9810801-3-1/DATE08 © 2008 EDAA

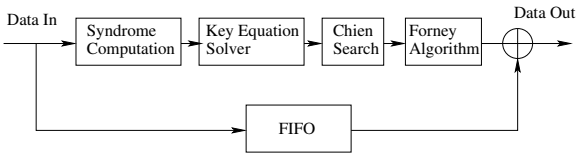

Figure 2. *RS decoder.*

module. This *key equation solver* module generates two polynomials for determining the location and value of errors in the received code word, if any. The next module called *Chien Search* computes the error locations, while the fourth module employs the Forney algorithm to determine the values of errors detected. A good summary of hardware resources needed for different algorithms for each module for ASIC implementation is provided in [3]. Interestingly, the hardware required is proportional to the error correction capability of the code and not the actual code word length as can be found in [1].

3. Embedded Vector Processor

The EVP_{16} (Embedded Vector Processor [4]) is a productized version of the CVP [5]. Although originally developed to support 3G standards, the current architecture proves to be more versatile. The architecture combines SIMD and VLIW parallelism as depicted in Figure 3. The main word width is 16 bits, with support for 8-bit and 32-bit data. The EVP_{16} supports multiple data types, including complex numbers. For example, a complex vector multiplication uses 16 multipliers to multiply 8 complex numbers in two clock cycles. EVP_{16} has 16 vector and 32 scalar registers. The maximum VLIW-parallelism available equals five vector operations *plus* four scalar operations *plus* three address updates *plus* loop-control.

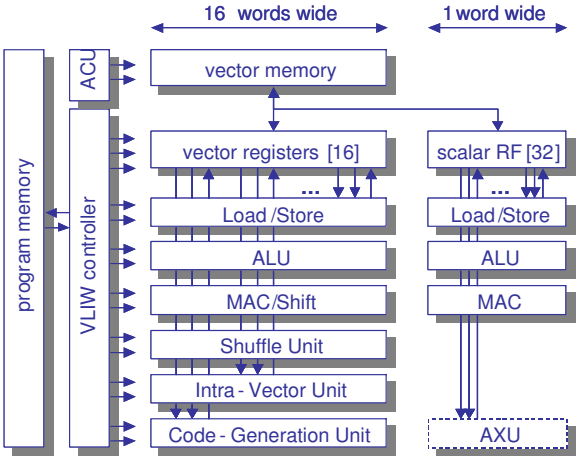

Figure 3. *The EVP-16 architecture*

In a 90 nm CMOS process, the EVP_{16} core measures about 3 mm^2 (600 k gates), runs 300 MHz (worst case commercial), and dissipates about 1 mW/MHz including a typical memory configuration.

Programs are written in EVP-C, a superset of ANSI-C. The extensions include vector data types and function intrinsics for vector operations, all in a C-like syntax. The EVP-C compiler takes care of register allocation (scalar and vector registers) as well as VLIW instruction scheduling (scalar and vector operations combined).

4. Vectorization of the RS Modules

Each of the following sub-Sections discusses the design decisions made when implementing RS decoder to EVP. The vectorization technique is different for each module and is determined by how the parallelism in EVP can be best exploited. Scalability is also discussed for each module. The vector processor used here uses vectors of 256 bits divided into 32 8-bit data elements.

4.1. Syndrome Computation

Syndrome computation is the first stage of RS decoding. $2t$ syndrome coefficients are needed for decoding a code word with error correction capability of t. Coefficients s_i, with $i = 1, 2, \ldots 2t$ are computed according to the following equation

$$s_i = \sum_{j=o}^{N-1} r_j(\alpha^i)^j \tag{2}$$

where r_j refers to N received symbols and α is a root of the primitive polynomial [1]. In order to reduce the computational requirement and complexity, syndromes are often computed using Horner's rule, according to the following recursive formula

$$s_{i,j} = s_{i,j-1}\alpha^i + r_{N-j}, j = 1, 2, \ldots N \tag{3}$$

where $s_{i,0}$ is set to 0 for each code word at the start of algorithm. At the end of algorithm, $s_{i,N}$ contains the required coefficient. Figure 4 shows an example of a syndrome computation cell that represents the above equation. The algorithm to compute syndromes using this formula is presented in Algorithm 1.

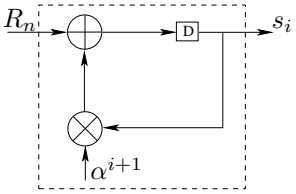

Figure 4. *A typical syndrome computation cell.*

Algorithm 1 Syndrome: Serial Computation

```
1: Read received symbols R[N]
2: for i = 1 to 2 × t do
3:     s_i = 0
4:     for j = 1 to N do
5:         s_i ← s_i × α^i + R[N − j]
6:     end for
7: end for
```

For RS (255, 239) code, 16 syndromes need to be computed. The simplest approach is to use a vector to store

16 syndrome values of 8-bits, and process one symbol at a time using Equation 3. However, this utilizes only half the vector space, and requires 255 iterations. In order to exploit the entire vector size, we modify the formula to divide each syndrome coefficient into two parts − one for received symbols at even locations and one at odd locations.

$$\begin{aligned} s_i &= (\ldots(r_{N-1}\alpha^i + r_{N-2})\alpha^i + \ldots + r_1)\alpha^i + r_0 \\ &= (\ldots(r_{N-1}\alpha^{2i} + r_{N-3})\alpha^{2i} + \ldots + r_1)\alpha^i + \\ &\quad (\ldots(r_{N-2}\alpha^{2i} + r_{N-4})\alpha^{2i} + \ldots + r_0) \\ &= (s_{i,odd})\alpha^i + (s_{i,even}) \end{aligned}$$

where $s_{i,even}$ is the summation of all symbols received at even locations, while $s_{i,odd}$ of those received at odd locations. (The code word may be padded with 0's to make N a multiple of vector-size. In RS(255,239), since N was 255, and vector-size 32, we added one 0 to make N equal to 256.) The two parts are later combined at the end of the algorithm. This approach requires only 128 iterations to evaluate the syndrome. Algorithm 2 shows the psuedo-code for the same.

Algorithm 2 Syndrome: Parallel Computation

```
Read received symbols rcvd[N]
Set syndrome vector, S to 0.
Set α_high = α_low = ⟨α_2, α_4, ... α_4t⟩.
for j = 1 to N/2 do
    Set R_high = ⟨R[N − 2j]⟩, R_low =⟨R[N − 2j + 1]⟩
    S ← S × α + R
end for
Set α_high = ⟨1, 1, ... 1⟩ and α_low = ⟨α_1, α_2, ... α_2t⟩.
S ← S × α
S2 ← S rotated by 16 symbols
S ← S + S2
S_high = S_low = required syndrome coefficients
```

Half Syndromes. It has been mentioned that $2t$ syndrome coefficients are needed for evaluating the correct symbols transmitted, in case of errors. However, if any t continuous syndromes are zero, we may conclude that the code word is either correct or uncorrectable [3]. In practice many of the code words are received correctly. Therefore, in our design we only compute half the syndromes i.e. 8 for RS (255, 239) and compute the rest only if any of these eight is non-zero. In order to exploit the full vector space of 32 elements in EVP, we now need four streams in parallel. The equation for these streams is derived as follows.

$$\begin{aligned} s_i &= (\ldots(r_{N-1}\alpha^i + r_{N-2})\alpha^i + \ldots + r_1)\alpha^i + r_0 \\ &= (\ldots(r_{N-1}\alpha^{4i} + r_{N-5})\alpha^{4i} + \ldots + r_3)\alpha^{3i} + \\ &\quad (\ldots(r_{N-2}\alpha^{4i} + r_{N-6})\alpha^{4i} + \ldots + r_2)\alpha^{2i} + \\ &\quad (\ldots(r_{N-3}\alpha^{4i} + r_{N-7})\alpha^{4i} + \ldots + r_1)\alpha^i + \\ &\quad (\ldots(r_{N-4}\alpha^{4i} + r_{N-8})\alpha^{4i} + \ldots + r_0) \\ &= (s_{i,1})\alpha^{3i} + (s_{i,2})\alpha^{2i} + (s_{i,3})\alpha^i + (s_{i,4}) \end{aligned}$$

Each vector therefore contains 4 values of each syndrome coefficient. Thus, in case of no errors, only 256/4 = 64 iterations are needed.

Scalability. The algorithm scales very well with the size of vector. When the vector size is doubled, the cycle count is almost halved as shown later in Section 5. For common code word sizes, the scalar overhead is small compared to the loop body, for vector sizes up to 64.

4.2. Key Equation Solver

A number of algorithms and architectures are available for solving the Key-equation [3]. Berlekamp Massey (BM) and Euclidean are the most common algorithms, and each has several architecture options for implementation. The algorithm or architecture that is most suited for ASIC design may not be ideal for implementation on a vector processor. Some of the characteristics that make an algorithm ideal for vectorization are

- Parallelism: This determines to what extent we can exploit the length of vector and the benefits of using vector processor. A serial implementation of BM algorithm for example has very little SIMD parallelism and hence defeats the purpose of using a vector processor.
- Regularity: Leads to shorter code since loops can be used, and hence less program memory requirements.

In view of the above concerns, the dual-line architecture of BM algorithm is selected for implementation. Unfortunately, the full parallelism in the vector processor still can not be fully exploited, as we only need vector of size $2 \times t = 16$ for RS (255, 239), while our vector was of size 32. There is no algorithm that allows the use of full capability of our vector processor. Figure 5 shows the architecture of the dual-line implementation.

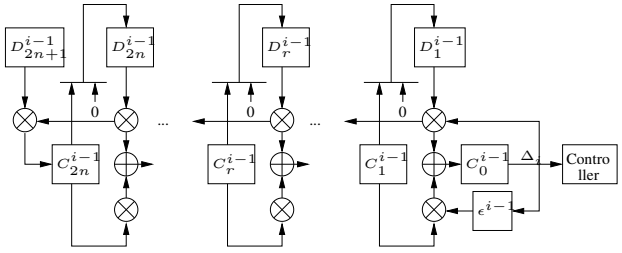

Figure 5. *Dual-line architecture for modified BM.*

Scalability. This algorithm doesn't scale very well, since it only allows exploiting parallelism in processors of size up to $2 \times t$. For RS(255, 239), with an error correction capability of 8, vector of size 16 is enough. The dependency between different cycles prohibits further exploitation of the EVP. As we see in Section 5, the instruction count for Key Equation solver block remains the same (above parallelism of 16), regardless of the level of parallelism in the processor.

4.3. Chien Search

The Chien module is trivial to parallelize and scales well with any level of parallelism available in the processor. This module checks if there is an error at a given location. Various positions can be evaluated in parallel, since there is little dependency. There is some overhead as compared to a fully scalar design, but not significant when compared to the gain achieved by vectorization.

4.4. Forney Algorithm

The Forney module is the last stage of the decoder and computes the value of the error at the computed error locations. This value is then subtracted to obtain the actual symbol. One difference in this module from the rest of the decoder is the data dependency of the algorithm. This modules computes the values at locations where error has indeed occurred; and that information is available after computation of Chien. Further, this part of decoder requires a division (or an inverter and a multiplier) which is fairly expensive. There are three ways to implement a divider:

- Hardware: A dedicated divider in hardware is not desirable, since it is only used in this module and it is fairly expensive in silicon area.
- Lookup-table: Another way is to store the inverse in a look-up table and then multiply the dividend with the inverse of the divisor. The size of lookup table is 256 bytes. A 32-way lookup table requires 8 kB ($32 \times 256B$), which is also excessive given its limited use.
- Micro-code: Since we are working in galois arithmetic, computing the inverse of an element or x^{-1} is equal to computing x^{N-1}. In our case $N = 255$, so we need to compute x^{254}, which can be computed using galois multipliers. It takes 11 multiplications to compute the inverse, thereby requiring 12 multiplications to complete the division. Since we have 255 symbols, and 32 are processed at one time, we only need eight iterations for the entire code word. The number of instructions needed is not very high and therefore feasible. In the final implementation, the micro-code is used to implement a divider.

Another decision to be taken in designing algorithm for this module is whether to compute the error values only at the error locations, or for the entire code word. The former ("selective error computation") requires the extraction of all error locations from the code word, which can hardly be accelerated by means of SIMD instructions. The latter ("oblivious error computation") can be refined, by computing error values only for those vectors that contain an error location. The choice between selective and oblivious error computation depends on the capabilities of the vector processor, the code size, and on the number of errors. For the benchmarked cases, oblivious error computation is faster.

Scalability. The actual algorithm scales gracefully with the length of vector. When the length of vector is doubled, the number of instructions is almost exactly halved. However, the choice of algorithm and architecture is dependent upon the vector-size. If the size of vector is small, a look-up table may be more feasible than using a micro-code.

4.5. Galois Multiplier

Galois multiplier forms an integral part of every module in RS decoding, and is therefore discussed here separately. There are various ways to implement Galois multiplier in the EVP architecture. Some of them are mentioned below and their trade-offs are discussed.

- Pure Hardware: An 8-bit galois multiplier requires about 140 gates in CMOS [6]. For 32 multiplications in parallel, we therefore need about 4K gates in total. The total number of gates in EVP is around 600K, which puts the cost of parallel Galois multiplication to less than 1% of the total chip.
- Look-up Table: Since we have two 8-bit multiplicands, using a look-up table requires storing 16-bit values and 8-bit result. This translates to a table of size 64 KB for each multiplication. Since we need 32 of these in parallel, we need a table of size $32 \times 64KB$ i.e. 2 MB.
- Hybrid: This approach uses the property that multiplication is equivalent to addition in logarithmic domain. The logarithm of the multiplicands is *looked-up* in a log table which is considerably smaller than the first case, since we only need to store 8-bit values. The two log values are added and the result is found by another lookup in the alog table. For Galois arithmetic, the base for both the tables is α. Though it takes considerably less memory, it comes at the cost of extra clock cycles needed for it. If we use one log table (for each multiplication), we need 4 cycles and if we use two tables, we require 3 cycles. In the former case we need 16KB of memory while the latter requires 24 KB. Another very big advantage of this method is the ease with which division can be implemented. Since division is simply subtraction in log domain, the same trick can be used to implement division with the same hardware.
- Pure Software: This approach is analogous to using micro-code for implementing complicated instructions. The psuedo-code is shown in Algorithm 3. However, this takes about 40 instructions and thus renders the implementation infeasible.

Algorithm 3 Galois Multiplication

```
1: Read 8-bit inputs a and b
2: Set product, p to 0; p ← 0
3: for i = 0 to 7 do
4:    if b(0) = 0 then
5:       p ← p ⊕ a
6:    end if
7:    c ← a(7)
8:    Shift a one bit to left
9:    if c = 0 then
10:      a ← a ⊕ 0x1b
11:   end if
12:   Shift b one bit to right
13: end for
14: p has the final product
```

5. Results and Benchmarking

5.1. EVP results

For the EVP, it is assumed that a hardware galois multiplier is present, unless otherwise mentioned. Since for our example we use 8-bit symbols, the supported vector size equals 32. RS (255, 239) was used for benchmarking. Since EVP-C is C-based, programming the RS modules was relatively easy. Ten thousand test cases were tested in all for

Table 1. Cycle count of various modules for EVP implementation

		Constant	Loop	Count	Total*	Scalar	Vector	S/w Division	Multiplications
Syndrome	1st half	12	4	15	72	6	66	66	66
	2nd half	20	4	15	80	12	68	68	68
Key Equation	EEP	18	6	15	108	18	90	90	32
	ELP	6	3	9	33	6	27	27	9
	Total				141	24	117	117	41
Chien		7	5	4	27				
			27	8	216	6	210	210	128
Forney		10	2	7	24				
		6	24	8	198	6	192	272	208
Total					707	54	653	733	511

*Total = Constant + (Loop $\times$ Count)

Table 2. Cycle count of various modules for different architectures

	EVP*, RS(255, 239)				TI DSP		Starcore
	H/w Divider	S/w Divider			RS(208, 188)	RS(255, 239)	RS(255, 239)
Parallelism Degree	32	32	16	64	8	8	1
Syndrome	72	72	138	39	470	576	5800
	80	80	148	51			
Key Equation	141	141	141	141	263	263	3816
Chien	216	216	426	111	326	399	4128
Forney	198	278	550	142	154	188	590
Total Number of Cycles	707	787	1403	484	1213	1426	14334
Worst Case Throughput (in Mbps)	860	778	440	1264	411	430	43
Best Case Throughput (in Mbps)	8500	8500	4430	15700	1060	1060	105

different number of symbol errors in the code word. The errors in the test cases were introduced randomly.

We first verified the functional correctness of the decoder by implementing the galois multiplier using a look-up table, and later used a dummy vector instruction to test the instruction count of the algorithm. The initial count obtained was to the tune of 3000 instructions for decoding one code word of 255 symbols i.e. 2040 bits. This was optimized in various iterations after discussions from the compiler experts. Some of the optimizations that helped significantly in reducing the cycle count are listed below.

- *Shuffle instruction:* The EVP shuffle instruction allows selective copying of content from one vector to another based on the pattern provided in a third vector.
- *Conditional branches:* Some conditional branches could be replaced by cheaper *masked instructions*.
- *Hardware-loop support:* Loops with a fixed number of iterations run faster by using hardware loop support. The counter-type was modified such that the EVP-C compiler recognizes these loops.
- *Multiple Streams:* To exploit the full VLIW parallelism, multiple streams for the Syndrome computation were introduced. Using four streams is optimal, since there are four instructions in each loop.

With the above and some other minor optimizations, the instruction count was reduced to about 700 for decoding in the worst case (8 errors). For the best case (no errors) the instruction count was reduced to as low as 80 instructions. Table 1 shows the number of instructions needed for the different modules. The column labeled 'Constant' shows the number of constant instructions for that module, while 'Loop' is the number of instructions in a loop and the next column indicates the number of iterations of that loop. Columns 'Scalar' and 'Vector' indicate the number of respective instructions in the particular module. The second last column is an estimate of how many vector instructions are needed when galois division is carried out in software, and the last column gives the number of galois multiplication operations in the entire decoding process.

As expected, software division affects only the Forney module, as it is the only one which needs galois division. We also observe that the majority of operations are vector operations. In some modules where we have two loops – inner and outer loops – the first row shows the inner loop and the second one shows the outer loop.

5.2. Comparison with other implementations

Table 2 shows the cycle counts needed in RS decoding for various architectures. We have compared our implementation with two other vector processors - TI [7] and Starcore [8][1]. For the TI DSP, we upscale the results for RS (255, 239) for a fair comparison. We also estimate how many cycles are needed when the vector parallelism in EVP is changed to 16 or 64. In the second column, we show results with a galois divider in hardware, while the other columns assume that the division is done in software. The throughput achieved for various architectures is also shown. The current EVP is capable of running at 300 MHz. Thus, the maximum throughput which can be obtained corresponds to the case when no errors are found i.e. about 8.5 Gbps, while the worst case throughput is about 800 Mbps. In comparison, assuming same frequency, TI achieves a best case throughput of 1.1 Gbps while the worst case throughput of 430 Mbps. For Starcore, the best and worst case throughput is 400 and 40 Mbps respectively at the same frequency.

Figure 6 shows an estimate of cycle count expected for different bit error rates (*BER*) in transmission[2]. As mentioned earlier, in the case of no errors, we only need to

[1] Reconfigurable architectures have also been used for implementation, e.g. Morphosys [9]. However, their area and power consumption often makes them undesirable. Morphosys has ten times as much area (with technology scaling) as EVP and consumes thirteen times as much power.

[2] Analysis of BER versus SNR can be found in [10].

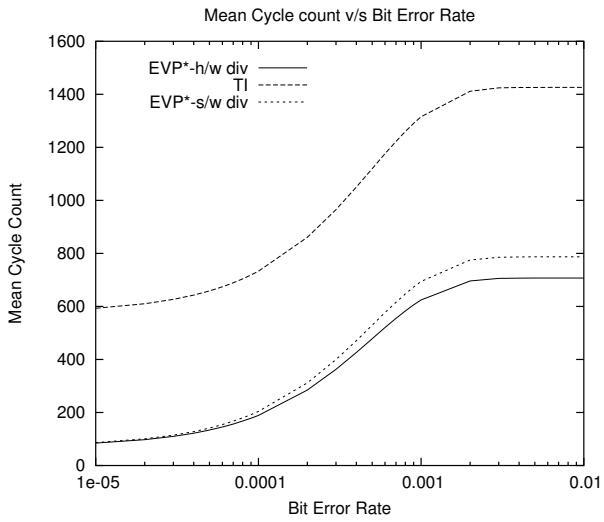

Figure 6. Mean cycle count v/s bit error rate

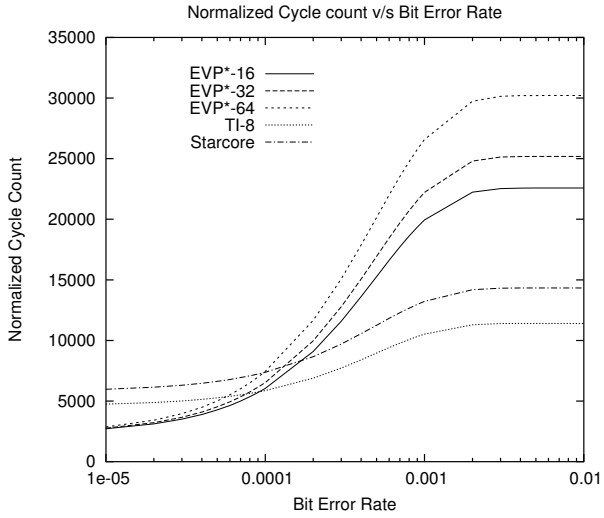

Figure 7. Normalized cycle count v/s bit error rate

compute half the syndromes. Therefore, as the BER decreases to below 0.0001, the number of cycles needed drops to about 80 for EVP and about 600 for TI. The reason for lower cycle count in EVP is increased parallelism and computation of only half the syndromes. With increasing BER, the number of packets with errors increases as well. When the BER rises over 0.001, most of the packets have error(s) and the cycle count approaches the worst case. The EVP takes about half as many cycles as needed by TI, instead of a quarter, because of scalar instructions which limit the exploitation of full parallelism in EVP.

Figure 7 shows the cycle count normalized with the amount of parallelism present in the architecture for varying BER. The normalized cycle count is obtained by multiplying the cycle count with vector parallelism in the architecture. Thus, TI is multiplied by 8, and EVP is multiplied by 16, 32, or 64 depending on the corresponding design. Starcore operates on a single symbol and was therefore, not scaled up. We again observe that at lower BER, the mean cycle count is lower and corresponds to the number of cycles needed to detect if there is any error in the received code word. As expected, the number of cycles for EVP is only half as compared to the rest since only half the syndromes are computed. Since syndrome computation module scales gracefully, high parallelism does not cause any inefficiency in this module. Another observation we make is that high parallelism is not very efficient. This comes from the fact that not all blocks of the algorithm scale ideally with increasing parallelism. Key Equation solver in particular scales only to a maximum parallelism of 16, and significantly decreases the efficiency for architectures with higher parallelism. However, if decoding speed is crucial, higher parallelism gives higher throughput.

6. Conclusions

In this paper, we present various vectorization techniques for the modules in Reed Solomon decoding. We see that different modules need to be vectorized in different ways. The vectorized decoder is implemented on the EVP and the results are compared with other SIMD implementations of RS decoder. The best and worst case throughput of our implementation is about 8.5 Gbps and 800 Mbps respectively. The best case results are about 8 times higher than the other implementations. However, we observe that there is no architecture that suits all scenarios and the desired architecture should be chosen depending on the decoding speed requirement and bit error rates of the transmission channel. Future research will focus on optimizing power.

Acknowledgements

We would like to thank Wim Kloosterhuis and Mahima Smriti from NXP's DSP Innovation Center for their help in optimizing the EVP-C code.

References

[1] S. B. Wicker and V. K. Bhargava; *Reed Solomon Codes and Their Applications*, Piscataway, NJ: IEEE Press, 1994.

[2] R. E. Blahut; *Theory and Practice of Error Control Codes*, Addison-Wesley, 1983.

[3] A. Kumar and S. Sawitzki; *High-Throughput and Low-Power Architectures for Reed Solomon Decoder*, Asilomar Conference on Signals, Systems and Computers, 2005.

[4] C.H. (Kees) van Berkel et al; *Vector Processing as an enabler for Software-Defined Radio in Handheld Devices*, EURASIP Journal on Applied Signal Processing, 2005.

[5] C.H. (Kees) van Berkel et al; *CVP: A Programmable Co Vector Processor for 3G Mobile Base-band Processing*, Proc. World Wireless Congress, May 2003.

[6] C. Paar and M. Rosner; *Comparison of arithmetic architectures for Reed-Solomon decoders in reconfigurable hardware*, IEEE Symposium on FPGAs for Custom Computing Machines, 1997.

[7] J. Sankaran; *Reed Solomon Decoder: TMS320C64x Implementation*, Texas Instruments Inc, SPRA686, Dec. 2000.

[8] D. Taipale et al; *Reed Solomon Decoding on the StarCore Processor*, Motorola Semiconductors Inc, AN1841/D, May 2000.

[9] A. Koohi, N. Bagherzadeg and C. Pan; *A fast parallel Reed-Solomon decoder on a reconfigurable architecture*, CODES + ISSS, Oct. 2003.

[10] A. Kumar; *High-throughput Reed-Solomon decoder for Ultra Wide band*. Masters Thesis, 2004, http://www.ics.ele.tue.nl/∼akash.