

Optimal Finite Field Multipliers for FPGAs

Gregory C. Ahlquist, Brent Nelson, and Michael Rice

459 Clyde Building, Brigham Young University
Provo UT 84602 USA

{ahlquist,nelson,mdr}@ee.byu.edu

Abstract. With the end goal of implementing optimal Reed-Solomon error control decoders on FPGAs, we characterize the FPGA performance of several finite field multiplier designs reported in the literature. We discover that finite field multipliers optimized for VLSI implementation are not optimized for FPGA implementation. Based on this observation, we discuss the relative merits of each multiplier design and show why each does not perform well on FPGAs. We then suggest how to improve the performance of many finite field multiplier designs¹.

1 Introduction

Since its advent in 1960, efficient and practical Reed-Solomon (RS) decoder design has been an important research topic. RS decoders decode transmitted data using Reed-Solomon linear cyclic block codes to detect and correct errors. Applications include error correction in compact disk players, deep space communications, and military radios. Traditionally, RS decoder design exploits Application Specific Integrated Circuit (ASIC) technology to realize low processing times and small areas. Nevertheless, ASICs achieve these advantages at the expense of flexibility. General Purpose Processors (GPPs), Digital Signal Processors (DSPs), or hybrid ASICs can provide flexibility, but do so at the expense of processing time and area. FPGAs, however, promise to realize RS decoders with time and area performance similar to ASICs but with flexibility similar to GPPs. This is intriguing as it opens the door to dynamically changing error control coding based on the current characteristics of the transmission channel. Indeed, FPGA-hosted RS decoders already exist but little work has been reported on realizing FPGA-based dynamic error control or on simply characterizing the optimal FPGA implementation of RS decoders. Thus, we have focused our research in these areas. As finite field multipliers are the most numerous building blocks of RS decoders, our initial efforts seek to characterize the optimal FPGA finite field multiplier implementation. The rewards could be significant. We recently implemented a RS decoder for the RS(15,9) code on a Xilinx XC4062. This relatively small decoder required 130 finite field multipliers using 800 Configurable Logic Blocks (CLBs) and ran at 15 Megabits (Mb) per second. We estimate a larger decoder, such as for the industry standard 30 Mb RS(255,233) code [1],

¹ Distribution A: Approved for public release; distribution unlimited

would require approximately 15,000 multipliers using 270,000 CLBs; equivalent to 118 Xilinx XC4062 Chips! There are a number a RS decoder design parameters we can adjust to reduce the resource requirement, increase speed, and make FPGA-based finite field decoders more feasible. Nevertheless, attempting to optimizing FPGA-based finite field multipliers in terms of both speed and area is an important first step.

2 Finite Fields

Reed-Solomon codes are based on finite field arithmetic which involves defining closed binary operations over finite sets of elements. Unfortunately, a full review of finite fields is beyond the scope of this presentation but an excellent treatment of the topic is found in [2]. As a brief overview, we will start with the simplest example of a finite field which is the binary field consisting of the elements $\{0,1\}$. Traditionally referred to as $\text{GF}(2)$ ², the operations in this field are defined as integer addition and multiplication reduced modulo 2. We can create larger fields by extending $\text{GF}(2)$ into vector space leading to finite fields of size 2^m . These are simple extensions of the base field $\text{GF}(2)$ over m dimensions. The field $\text{GF}(2^m)$ is thus defined as a field with 2^m elements each of which is a binary m -tuple. Using this definition, we can group m bits of binary data and refer to it as an element of $\text{GF}(2^m)$. This in turn allows us to apply the associated mathematical operations of the field to encode and decode data. For our purposes, we will limit our discussion to the finite field $\text{GF}(16)$. This field consists of sixteen elements and two binary operations; addition and multiplication. There are two alternate (but equivalent) representations for the field elements. First, all nonzero elements in $\text{GF}(16)$ may be represented as powers of a primitive field element α [2] (i.e. each nonzero element is of the form α^n for $n = 0, 1, \dots, 14$). Second, each element has an equivalent representation as a binary 4-tuple. While the α^n representation has great mathematical convenience, digital hardware prefers the binary 4-tuple representation. These representations are illustrated in Table 1.

Table 1. Canonical Representation of Finite Field $\text{GF}(16)$

Element	0	α^0	α^1	α^2	α^3	α^4	α^5	α^6
Representation	0000	0001	0010	0100	1000	0011	0110	1100
Element	α^7	α^8	α^9	α^{10}	α^{11}	α^{12}	α^{13}	α^{14}
Representation	1011	0101	1010	0111	1110	1111	1101	1001

To understand how finite field addition and multiplication work, it is essential to view each field element as describing the coefficients of a binary polynomial.

² The GF stands for Galois Field which is another name for a finite field. The term honors the French mathematician Eviriste Galois who first formalized finite field concepts.

For example, element α^7 whose binary representation is 1011 represents the polynomial $\alpha^3 + \alpha^1 + \alpha^0$. You may gain added insight by noting that the element α^7 is in fact the linear summation of α^3 , α^1 , and α^0 . Under this representation, finite field addition and multiplication become polynomial addition and multiplication where the addition operation occurs modulo 2. To illustrate, consider GF(16) elements A and B represented as follows:

$$A = a_3\alpha^3 + a_2\alpha^2 + a_1\alpha^1 + a_0\alpha^0 \quad (1)$$

$$B = b_3\alpha^3 + b_2\alpha^2 + b_1\alpha^1 + b_0\alpha^0 \quad (2)$$

Adding elements A and B becomes a simple bit-wise modulo 2 addition.

$$A + B = (a_3 + b_3)\alpha^3 + (a_2 + b_2)\alpha^2 + (a_1 + b_1)\alpha^1 + (a_0 + b_0)\alpha^0. \quad (3)$$

Multiplication is a little more complicated. We begin by carrying out the polynomial multiplication.

$$\begin{aligned} A * B &= a_3 * b_3\alpha^6 + (a_3 * b_2 + a_2 * b_3)\alpha^5 \\ &\quad + (a_3 * b_1 + a_2 * b_2 + a_1 * b_3)\alpha^4 \\ &\quad + (a_3 * b_0 + a_2 * b_1 + a_1 * b_2 + a_0 * b_3)\alpha^3 \\ &\quad + (a_2 * b_0 + a_1 * b_1 + a_0 * b_2)\alpha^2 \\ &\quad + (a_1 * b_0 + a_0 * b_1)\alpha^1 + a_0 * b_0\alpha^0 \end{aligned} \quad (4)$$

The result has seven coefficients which we must convert back into a 4-tuple to achieve closure. We do this by substituting α^6 , α^5 , and α^4 with their polynomial representations and summing terms.

$$\begin{aligned} A * B &= (a_3 * b_3 + a_3 * b_0 + a_2 * b_1 + a_1 * b_2 + a_0 * b_3)\alpha^3 \\ &\quad + (a_3 * b_3 + a_3 * b_2 + a_2 * b_3 + a_2 * b_0 + a_1 * b_1 + a_0 * b_2)\alpha^2 \\ &\quad + (a_3 * b_2 + a_2 * b_3 + a_3 * b_1 + a_2 * b_2 + a_1 * b_3 + a_1 * b_0 + a_0 * b_1)\alpha^1 \\ &\quad + (a_3 * b_1 + a_2 * b_2 + a_1 * b_3 + a_0 * b_0)\alpha^0 \end{aligned} \quad (5)$$

Equation (5) is often expressed in matrix form.

$$\begin{pmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_3 + a_0 & a_3 + a_2 & a_1 + a_2 \\ a_2 & a_1 & a_3 + a_0 & a_3 + a_2 \\ a_3 & a_2 & a_1 & a_3 + a_0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} \quad (6)$$

Because GF(16) is an extension of the binary field, The multiplies in equation (5) can be implemented as logical ANDs and the additions as logical XORs.

Thus, the expression requires only sixteen ANDs and nine XORs to implement. Nevertheless, a number of hardware designs to perform finite field multiplication have been devised [3,4,5,6,7,8,9,10]. Each new design strives to reduce the overall silicon area and time required for the operation.

3 Finite Field Multipliers

The dependence on finite field arithmetic means that efficient RS decoders are dependent on efficient finite field adders and multipliers. Addition is easy. It equates to a bit-wise XOR of the m -tuple and is realized by an array of m XOR gates. The finite field multiplier is, by comparison, much more complicated and is the key to developing efficient finite field computational circuits. We have conducted an extensive survey of finite field multiplier designs and have characterized their performance on an FPGA. For each design, we created a hardware description using the Brigham Young University developed hardware description language JHDL [11]. This language easily allows us to model, simulate, and netlist our multiplier designs. Using JHDL, we verified the correct operation of each design and created an associated netlist. There were seven designs in all.

3.1 Linear Feedback Shift Register Multiplier

The Linear Feedback Shift Register (LFSR) GF(16) Multiplier design is considered by many the standard or canonical design for finite field multipliers. An excellent and detailed description of this multiplier can be found in [3]. The LFSR GF(16) multiplier has parallel input and output (i.e. inputs and outputs occur in 4-bit blocks) but possesses a 4 clock cycle latency which is not pipelined. Thus, the LFSR multiplier produces one 4-bit result every 4 clock cycles. As Figure 1 illustrates, the hardware consists of a latch and a shift register that feed arguments into a central LFSR. Appropriate combinational logic is appended to the LFSR to combine together the arguments, feedback, and the previous partial product.

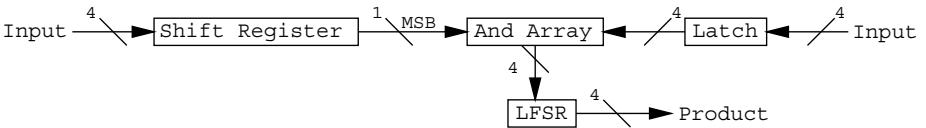


Fig. 1. Block Diagram of the Canonical LFSR Finite Field Multiplier.

3.2 Mastrovito Multiplier

In his doctoral dissertation, Mastrovito [4] provides a completely combinatorial finite field multiplier design. Mastrovito's multiplier has parallel inputs and out-

puts and no clock cycle latency. Thus, this multiplier produces one 4-bit result every clock cycle. Mastrovito's design mirrors the matrix form of the finite field multiplication equation (6) and consists of two stages. The first stage computes all the multi-variable A matrix elements and feeds these to the second stage. The second stage performs the matrix multiplication and outputs the product. All operations are performed completely in combinatorial logic.

3.3 Massey-Omura Multiplier

The Massey-Omura Multiplier as reported by Wang et. al. [5] operates on GF(16) elements represented in the normal basis. The normal basis merely uses a different binary representation for each GF(16) element. The benefit is that, under this basis, equation (5) becomes highly uniform and facilitates a regular VLSI design. The Massey-Omura multiplier has parallel inputs but produces the product serially. There is no clock cycle latency between argument presentation and results, but it does take 4 clock cycles to produce all 4 bits of the product. As Figure 2 shows, the multiplier loads the arguments into two cyclic shift registers. These, in turn, feed a block of combinatorial logic that produces one bit of the result. On each subsequent clock cycle, the register contents shift and the combinatorial logic produces another bit of the product.

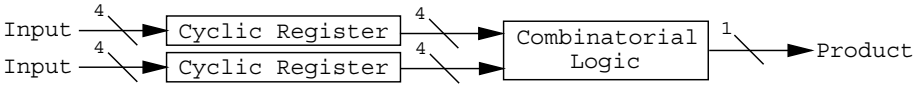


Fig. 2. Block Diagram of Massey-Omura Finite Field Multiplier.

3.4 Hasan-Bhargava Multiplier

Hasan and Bhargava [7] devised a bit-serial finite field multiplier implemented as a systolic array. The multiplier streams bits sequentially through eight small processing units with the product flowing bit-serially from the last processor. Each processing unit consists of two to three registers and a small amount of combinatorial logic. The first four processing units calculate the elements of the A matrix shown in equation (6). The second four units perform the matrix calculation. This design possesses an eight clock cycle latency and produces one 4-bit product every 4 clock cycles thereafter.

3.5 Paar-Rosner Multiplier

Paar and Rosner [8] present a novel multiplier design using a composite basis representation of GF(16). The composite basis essentially divides the GF(16)

element into upper and lower bit pairs where each pair represents an element of $GF(4)$. This allows Paar and Rosner to effect the multiplication of two $GF(16)$ elements by performing simpler $GF(4)$ multiplications and additions. The overall effect is to recast equation (5) in combinatorial logic in a very efficient manner. The Paar-Rosner multiplier has parallel inputs and outputs and no clock cycle latency.

3.6 Morii-Berlekamp Multiplier

A multiplier proposed by Morii et. al. [9] and based on a similar multiplier developed by Berlekamp [10] uses a dual basis representation of $GF(16)$ elements. The dual basis is again just another way to represent the elements of $GF(16)$ as binary 4-tuples. The advantage of using the dual basis is it converts the A matrix of equation (6) into a simpler form that can be realized by a simple LFSR. Like the Massey-Omura multiplier, the Morii-Berlekamp multiplier has parallel inputs but produces the output bit-serially. As Figure 3 shows, the multiplier loads one argument in a register and another into an LFSR. On each clock cycle, the LFSR computes a row of the simplified A matrix which is then fed into a block of combinatorial logic. The combinatorial logic block combines this input with the other argument to compute one bit of the result. Although there is no latency between argument presentation and results, the design does require 4 clock cycles to produce a complete 4-bit product.

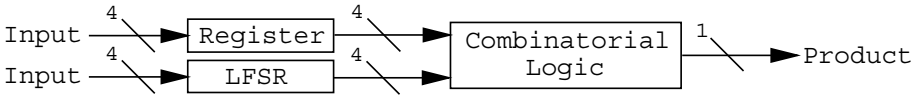


Fig. 3. Block Diagram of Morii-Berlekamp Finite Field Multiplier.

3.7 Pipelined Combinatorial Multiplier

This multiplier is a simple, pipelined implementation of equation (5) which we designed. The multiplier has parallel inputs and outputs and consists of two blocks of combinatorial logic separated by a bank of registers. The first combinatorial logic block breaks equation (5) into eleven parts representing eleven partial products. The register bank latches the partial products and feeds them into the second logic block which combines the partial products to obtain the desired result. The multiplier possesses an initial 2 clock cycle latency but produces a result every clock cycle thereafter. This design is, in effect, yet another recasting of equation (5)

4 Finite Field Multiplier FPGA Performance

Using simulation tools available in our lab, we characterized the performance of each finite field multiplier on the Xilinx XC4062 FPGA. We used two metrics for comparing these designs. The first metric is the area-time product combining the number of Xilinx CLBs and the critical path delay measured in nano-seconds (ns). As we ideally want our designs to take negligible resources and time, a lower number in the CLB*ns column represents better performance. The second metric is like the first except it reports results in units of CLB*ns². This simple change adds emphasis to faster designs. Table 2 shows resource usage and clock speed data for all designs. The time value indicates the delay of each multiplier. For example, The Massey-Omura multiplier produces a complete 4-bit result every 40.016 ns.

Table 2. Finite Field $GF(16)$ Multiplier Comparison Results

Design	CLBs	Time(ns)	CLB*ns	CLB*ns ²
Massey-Omura	15	40.016	600.240	24019.4
Hasan-Bhargava	19	24.952	474.088	11892.4
LFSR	13	26.092	339.196	8850.3
Morii-Berlekamp	13	23.780	309.140	7351.3
Paar-Rosner	5	12.318	61.590	758.7
Mastrovito	6	9.784	58.704	574.4
Pipelined Combinatorial	7	6.336	44.352	281.4

5 Performance Analysis

Our study of these multipliers leads us to an interesting observation: finite field multipliers optimized specifically for VLSI are not necessarily optimized for FPGAs. To illustrate our claim, compare results of the literature reported designs with our simple pipelined combinatorial implementation of a finite field multiplier. The simple, straightforward design performs much better on the Xilinx XC4062 FPGA than the more complicated VLSI-optimized designs. This result is very much counter-intuitive to our original predictions and we attempted to explain why. On closer inspection, we observed that the VLSI optimized designs all possess architectures that do not map well to the underlying XC4062 and lead to poor performance. Three significant flaws in comparison to the pipelined combinatorial design emerged.

- Multi-clock cycle operation
- Long unregistered datapaths
- Underutilized logic elements

Specifically, we claim the pipelined combinatorial design outperforms all other designs because it produces a result every clock cycle (after an initial 2 clock cycle latency), registers all logic element outputs, and maximizes the resources of each logic element. Producing a result every clock cycle and registering all logic element outputs leads to a pipeline like architecture which shortens the clock cycle and increases throughput. Maximizing logic element resources minimizes the overall number of logic elements required. The result is a fast, small, pipelined combinatorial circuit that outperforms all other designs in terms of CLB^*ns and CLB^*ns^2 . In comparison, LFSR, Wang, Hasan-Bhargava, and Morii all perform worse than pipelined combinatorial primarily because they require 4 clock cycles to produce a complete 4-bit result. LFSR and Wang also suffer because they have unregistered datapaths that span more than one CLB. This greatly increases the signal propagation delay which in turn increases clock cycle time. Hasan-Bhargava requires an inordinately large number of resources because its small processing elements underutilize the processing capability of the Xilinx CLBs. LFSR, Wang, and Morii suffer from this as well but to a lesser extent. Mastrovito and Paar-Rosner produce complete 4-bit results every clock cycle and make excellent use of available logic resources. Nevertheless, they possess long, unregistered datapaths which increase their processing time. Table 3 summarizes each multiplier’s detractions when compared to the pipelined combinatorial design.

Table 3. Finite Field $GF(16)$ Multiplier Architecture Comparison

Design	Multi-Cycle Processing	Unregistered Datapaths	Underutilized Logic Elements
Massey-Omura	✓	✓	✓
Hasan-Bhargava	✓	✓	✓
LFSR	✓	✓	✓
Morii-Berlekamp	✓	✓	✓
Paar-Rosner		✓	
Mastrovito		✓	
Pipelined Combinatorial			

6 Performance Improvements

Recognizing these performance limitations puts us in the position to improve the performance of each design relative to the Xilinx XC4062 FPGA. Thus, we modified each design along the following guidelines.

- Eliminate multi-clock cycle operation
- Eliminate unregistered datapaths
- Maximize use of each logic element

Our strategy to eliminate multi-clock cycle operation was to pipeline the effected designs such that, after some initial latency, the multipliers produced a complete 4-bit result every clock cycle. The strategy to eliminate unregistered datapaths simply involved registering every signal as it emerged from a CLB logic processing element. The strategy for maximizing the use of each logic element involved eliminating CLBs used simply to register signals or that computed very simple logic results. In most cases, these strategies proved to be at odds with one another and our endeavor quickly turned into a classic time versus area tradeoff. Eliminating multi-clock cycle operation and unregistered datapaths necessitated the use of more Xilinx CLBs; primarily due to the need for more registers. Conversely, attempts to maximize the logic processing capability of each Xilinx CLB often led to multi-level datapaths. In the end, each design modification became an exercise in decreasing processing time while minimizing resource growth. Table 4 shows the performance of the designs listed in table 2 after modifications. A quick comparison with table 2 shows that we achieved some improvement in all designs with the exception of the Mastrovito multiplier. In the Mastrovito multiplier, one block of combinatorial logic feeds another and results in several signals spanning more than one CLB. In an attempt to reduce the length of these datapaths we registered all results from the first block. This introduced a timing problem which required us to register both 4-bit arguments; something we previously did not have to do. Although we were able to shorten the processing time for this multiplier, the associated cost in CLB resources due to the increased number of registers wiped out any gain.

We note with some emphasis that Paar-Rosner-II and Morii-Berlekamp-II either approach or slightly exceed the performance of our pipelined combinatorial design. We achieved these gains by breaking down the Paar-Rosner and Morii-Berlekamp designs into their basic governing logic equations and implementing them in the same fashion as our combinatorial pipelined design. The results imply that the Paar-Rosner and Morii-Berlekamp approaches are just as good as our pipelined combinatorial multiplier design so long as the respective multipliers are implemented in a pipelined combinatorial fashion.

Table 4. Modified Finite Field $GF(16)$ Multiplier Comparison Results

Design	CLBs	Time(ns)	CLB*ns	CLB*ns ²
Hasan-Bhargava-II	17	6.457	109.769	708.8
LFSR-II	17	5.981	101.677	608.1
Mastrovito-II	12	6.481	77.772	504.0
Massey-Omura-II	8	6.723	53.784	361.6
Morii-Berlekamp-II	7	6.338	44.366	281.2
Pipelined Combinatorial	7	6.336	44.352	281.0
Paar-Rosner-II	7	6.258	43.806	274.1

7 Conclusions

Our experiments with finite field $GF(16)$ multipliers taught us the following lessons. First, we cannot simply implement any finite field multiplier design reported in the literature and expect optimal FPGA performance. We must take into account the underlying architecture of the target FPGA. Specifically, optimal performance is dependent on single clock cycle throughput, registered datapaths, and maximum use of each logic element. Applying these principles to optimize finite field multiplier designs is a classic time-area tradeoff. We must be careful that design changes that decrease processing time do not increase resource usage such that any gains are nullified. Our data suggests that the optimal design is a simple, pipelined, combinatorial architecture that implements the governing logic equation of the multiplier. This is true regardless of the basis used to represent the elements of $GF(16)$. Our data also raises a number of unanswered questions. Do these conclusions hold true as the size of the multiplier grows to handle elements of $GF(32)$ on up to $GF(256)$? How much impact does the multiplier size and speed have on the overall design of RS encoders and decoders? How can we better design FPGAs to facilitate these kind of devices? Our future research will endeavor to address these and other questions.

References

1. J. L. Politano and D. Deprey. A 30 mbits/s (255,223) Reed-Solomon decoder. In *EUROCODE 90, International Symposium on Coding Theory and Applications*, pages 385–391. EUROCODE 90, Udine Italy, Nov 1990. 51
2. S. B. Wicker. *Error Control Systems for Digital Communication and Storage*. Prentice Hall, 1995. 52
3. S. Lin and D. J. Costello. *Error Control Coding: Fundamentals and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1983. 54
4. E. D. Mastrovito. *VLSI Architectures for Computations in Galois Fields*. PhD thesis, Linköping University, Dept. Electr. Eng., Linköping, Sweden, 1991. 54
5. C. A. Wang, T. K. Truong, H. M. Shao, L. J. Deutsch, J. K. Omura, and I. S. Reed. VLSI architectures for computing multiplications and inverses in $GF(2^m)$. *IEEE Transactions on Computers*, 34(8):709–717, Aug 1985. 54, 55
6. J. L. Massey and J. K. Omura. Computational method and apparatus for finite field arithmetic. U. S. Patent Application, 1981. 54
7. M. A. Hasan and V. K. Bhargava. Bit-serial systolic divider and multiplier for finite fields $GF(2^m)$. *IEEE Transactions on Computers*, 41(8):972–980, Aug 1992. 54, 55
8. C. Paar and M. Rosner. Comparison of arithmetic architectures for Reed-Solomon decoders in reconfigurable hardware. *IEEE Transactions on Computers*, 41(8):219–224, Aug 1997. 54, 55
9. M. Morii, M. Kasahara, and D. Whiting. Efficient bit-serial multiplication and the discrete-time Wiener-Hopf equation over finite fields. *IEEE Transactions on Information Theory*, 35(6):1177–1183, November 1989. 54, 56

10. E. Berlekamp. Bit-serial Reed-Solomon encoders. *IEEE Transactions on Information Theory*, IT-28(6):869–874, November 1982. 54, 56
11. B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting. A CAD suite for high-performance FPGA design. In K. Pocek and J. Arnold, editors, *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 99)*, page TBA. IEEE Computer Society, IEEE Computer, April 1999. 54