Automatic Generation of Polynomial-Basis Multipliers in $GF(2^n)$ using Recursive VHDL

J. Nelson, G. Lai, A. Tenca

***Abstract*— Multiplication in $GF(2^n)$ is very commonly used in the fields of cryptography and error correcting codes. Automating the design process for these multipliers could reduce the cost and development time of hardware implementations. In this project, we present a general design for these multipliers and a strategy to generate them automatically given the precision of the operands and the irreducible polynomial which defines the field. In particular, the generation and use of tree structures based on a previously proposed recursive VHDL technique is compared with other Galios Field multipliers. It is shown that the final result of this automatic design tool is very competitive with some specialized designs presented in the literature, with the advantage that it can be adjusted to any type of trinomial or pentanomial in the IEEE standard.**

***Index Terms*— recursive VHDL, $GF(2^n)$ multiplication, automatic generation, cryptography, reconfigurable.**

I. Introduction

MULTIPLICATION in binary extension fields, or Galois Fields $GF(2^n)$, is a very important operation in several numerical algorithms used in cryptographic applications and error-correcting codes [1]. The field is defined based on an irreducible polynomial $p(x)$ and as a result, the multipliers have a structure that depends on this polynomial [2]. Polynomials with fewer coefficients have less complexity, and for this reason the most common irreducible polynomials are trinomials and pentanomials (3 and 5 coefficients respectively). The IEEE standard shows a huge list of such polynomials [3]. Most of the work presented in the literature estimates the complexity of multipliers in $GF(2^n)$ [4], [5], [6], [7] and usually proposes design alternatives to optimize them. However, the actual design process, even with these design guidelines, is always a complex and tedious task. In this work we present an HDL (hardware description language) description for multipliers in $GF(2^n)$ that use polynomial basis. This description allows the generation of multipliers for any field size (within the limits of the synthesis tool) and for any trinomial or pentanomial. However, the basic concept is general and could be used for any type of irreducible polynomial. As in other multipliers, the fast addition of partial products is the most important factor in determining performance. The use of tree structures is the fastest way to perform this task. However, a parameterizable description of tree structures using VHDL can only be obtained when using recursive calls. The use of recursive VHDL is briefly explained in Section IV. The benefits of using an automatic generator of multipliers are flexibility and speed. From the information about the field size and possibly the irreducible polynomial, the multiplier is generated with minimal user intervention. This approach significantly reduces the cost of getting a working and efficient design solution. We initially present an overview of multiplication in $GF(2^n)$ and show the general design approach to be used in this work. A brief description of recursive VHDL is also shown. Some experimental results are presented and compared against published results. We initially imagined that the circuit would be inefficient, but the results show that the synthesized design is very competitive with optimized multiplier designs. These results are shown in later sections.

Authors are with the Department of Electrical Engineering & Computer Science, Oregon State University, Corvallis, Oregon 97331. E-mail: {nelsonja,laige,tenca}@ece.orst.edu

II. Polynomial Basis Multiplication in $GF(2^n)$

For our design we chose to use the standard or polynomial basis to represents elements in $GF(2^n)$. When using polynomial basis, each element of the field is represented by a polynomial of the form $a(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2}... + a_2x^2 + a_1x + a_0$ whose coefficients are always 1 or 0. To create a proper field an irreducible polynomial $p(x)$ of degree n must be chosen to define that field. All operations within the field are then performed modulo the polynomial $p(x)$.

Addition and subtraction, when using polynomial basis, is a simple XOR of the corresponding coefficients. Multiplication on the other hand is a very complicated procedure requiring the result to be reduced by $p(x)$ as shown in (1).

$$c(x) = a(x)b(x) \mod p(x) \tag{1}$$

This reduction can be performed by subtracting $p(x)x^i$ for each coefficient i whose degree is greater than $n-1$. If we view the multiplication as shown in (2) then it is possible to reduce each partial product seperately as shown in (3).

$$(a(x)b_{n-1}x^{n-1} + \cdots + a(x)xb_1 + a(x)b_0) \mod p(x) \tag{2}$$

$$c(x) = \sum_{i=0}^{n-1} a(x)b_ix^i \mod p(x) \tag{3}$$

In fact, the reduction can be further simplified by using each reduced partial product as the base for the next partial product. We can produce any partial product $z(x)_i$ as

$$z(x)_i = z(x)_{i-1}x + p(x)z_{n-1} \tag{4}$$

where z_{n-1} is the MSB of $z(x)_{i-1}$ and $z(x)_0 = a(x)$. Each $z(x)_i$ is then ANDed with b_i to select the actual partial products. This allows us to perform the reduction while calculating each partial product by adding $p(x)z_{n-1}$ at each iteration.

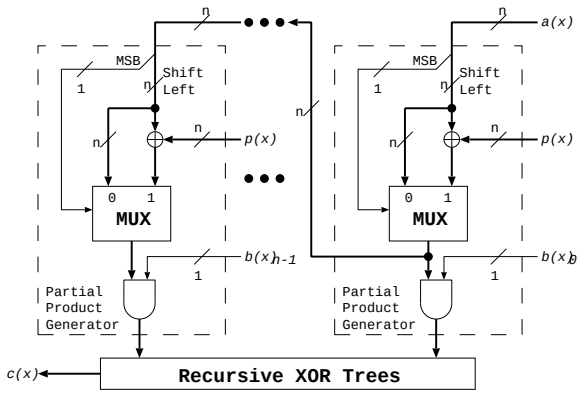

Fig. 1. Block Diagram of recursive implementation. Note that the result of each partial product module is fed into the next.

It is also possible to view the polynomial basis multiplication as the multiplication of a binary matrix and a binary vector as presented in [2]. Each column of the binary matrix contain one partial product $a(x)x^i$. The binary vector contains the coefficients of $b(x)$. It is then possible to create a reduction matrix Q based on $p(x)$. It has been shown that the logic required to add in the reduction matrix Q can be greatly simplified for most polynomials [4], [5], [6]. For all polynomials presented in [3] the Q matrix will be very sparse. This leads to a faster and smaller implementation.

III. Multiplier Design Approach

The multiplier we present in this paper is a Galois Field multiplier that operates on field elements defined using polynomial basis. The irreducible polynomial $p(x)$ can be prespecified for each desired multiplier design or one will be automatically chosen from [3] based on the operand size.

Rather than describing the matrix presented in [2] our HDL describes a structure with two main parts. The first is the the partial product generation stage and the second combines the resulting partial products. The combination of the partial products is performed by the recursive XOR trees described in Section IV.

Individual partial products are computed by left-shifting the previous partial product by one bit and checking the most-significant bit (MSB) of the result. If this bit is equal to 0, then the current partial product has not exceeded the degree of the field. If it is equal to 1, then it has exceeded the degree of field and must be reduced by $p(x)$. The reduction is accomplished by a bitwise XOR of the shifted partial product and $p(x)$. This way, the partial product computed in any iteration can always be represented in an n-bit vector. The computed partial product is passed on to the next iteration and this process continues until all partial products have been generated.

As shown in Figure 1, each dotted block represents one iteration step. In each block, the current partial product is left-shifted and its MSB is used to determine whether the partial product will be reduced by $p(x)$. The resulting partial product is then used in the next iteration step. Then, effective partial products are selected by AND operations of the partial product at block i with $c(x)_i$. To obtain the result, a bitwise-XOR of all the effective partial products is accomplished by means of an XOR tree.

IV. Recursive VHDL Description of Trees

Several efficient hardware circuits are constructed using trees. Describing these trees using non-recursive constructs can be very difficult. By using recursive VHDL [8], we found a compact and regular way to describe these trees. The following code is the architecture body of the entity called tree1:

```
BEGIN // beginning of architecture body
 degenerated_tree: IF d = 0 GENERATE
  output <= input(0);
 END GENERATE degenerated_tree;
 single_gate: IF d = 1 GENERATE
  xor_gate_root: xor_n
   GENERIC MAP(n => n)
   PORT MAP(a => input, z => output);
 END GENERATE single_gate;
 compound_tree: IF d > 1 GENERATE
  the_gate: xor_n GENERIC MAP(n => n)
   PORT MAP(a => xor_input,
    z => output);
  subtree_array: FOR i IN 0 TO n - 1
  GENERATE
   the_subtree: tree1
    GENERIC MAP(d => d-1, n => n)
    PORT MAP(input => input(i*n**(d-1)
     to (i+1)*n**(d-1)-1),
    output => xor_input(i));
  END GENERATE subtree_array;
 END GENERATE compound_tree;
END; // end of architecture body
```

where xor_n is an n-input XOR gate component. The variable d corresponds to the tree level, and depending on the level different instantiations are made. When $d = 0$ the tree is a single wire (degenerated tree). When $d = 1$ the level consists of a single XOR gate (the base of the tree). When the $d > 1$, one XOR gate is used, and each gate input is connected to one new tree. That is when the same component tree1 is invoked again. This VHDL code is extremely simple and flexible enough to generate trees of different sizes. It allows the control of the fan-in of the XOR gate and the number of tree levels. Once a tree is used to assimilate several bits from the partial products, there will most frequently be unused inputs. It is expected that the synthesis tool will prune unused logic from the tree generated from the recursive description. Given the sparse nature of the large penta- or trinomial, the synthesis tool is able to remove a large number of XOR gates that have all the inputs as zero, or only one variable with all the remaining inputs as zeros.

V. Initial Synthesis Results

To determine the characteristics of the proposed multiplier design, it was synthesized using several different operand sizes. For each run, one of the irreducible polynomials supplied in [3] were used. These polynomials included both

trinomials and pentanomials. The synthesis was performed using the Leonardo and the HEP ADK with its AMI-05 CMOS libraries.

As explained before, we expected to have the synthesis tool prune unused XOR gates from the tree. However, we did not expect that the synthesis tool would break the connections between partial product generators and make the final bit-slices largely independent. In doing this, the synthesis tool undoubtedly improved the delay of the final circuit.

Table I contains results from synthesis runs at 4, 8, 16, 32, 64, and 128 bits. We had attempted to synthesize larger versions but the system requirements exceeded the available resources. Ideally, we would like to determine the behavior of the recursive design at 512 and 1024 bits.

Table I
Auto-Generation ASIC Results

Size	XOR gates	AND gates	Delay
4-bit	15	16	$T_A + 3T_X$
8-bit	79	64	$T_A + 7T_X$
16-bit	287	256	$T_A + 7T_X$
32-bit	1,083	1,024	$T_A + 9T_X$
64-bit	4,221	4,096	$T_A + 10T_X$
128-bit	16,638	16,384	$T_A + 10T_X$

The memory requirement to synthesize designs with a large number of bits is significantly high. This is due largely to the number of instances need for such a large bit-parallel multiplier. The amount of memory available to the synthesis tool will almost certainly define the largest multiplier which can be created using auto-generation.

As can be seen, the area of the recursive design is quite large. Given the large number of XOR gates needed to construct the recursive trees, the area results are not surprising. It is interesting to note the delay characteristics of this design. Given the design of the partial product generation it was expected that there would be a large linear component, with a logarithmic component contributed by the XOR trees. However, when the synthesis tool optimized the partial product generation module, it changed the delay characteristics of the proposed multiplier structure.

VI. Comparison to Other Multipliers

For comparison, other parallel implementations of $GF(2^n)$ multipliers are introduced. The first is based on a modified version of the Karatsuba algorithm for polynomial multiplication. The second is a form of Mastrovito multiplier. For more details on these designs see [7,8]. The multipliers are similar to the one presented in this work in the sense that their delay grows logarithmically as n increases. It should be noted that the delays for the multipliers from [7,8] are purely theoretical and the delay and area characteristics are based solely on the algorithms. See Tables II and III for the estimated delay and area values.

The data shows that the recursive design is generally about twice as large as the Karatsuba multiplier. This is consistent for all the values of n we have synthesized. These results may indicate that there is more which can be done to optimize the area of the auto-generated multiplier. It is in the delay performance that the auto-generated design begins to show a real benefit. Figure 2 compares the delay characteristics of the recursive design with those of the other multipliers. As can be seen in the chart, the auto-generated design is able to out perform the Karatsuba multiplier at higher operand sizes.

Table II
Karatsuba-Ofman[7] ASIC Estimates

Size	XOR gates	AND gates	Delay
4-bit	9	16	$T_A + 2T_X$
8-bit	55	48	$T_A + 6T_X$
16-bit	225	144	$T_A + 10T_X$
32-bit	799	432	$T_A + 14T_X$
64-bit	2,649	1,296	$T_A + 18T_X$
128-bit	8,455	3,888	$T_A + 22T_X$

Table III
Mastrovito Multiplier[2] ASIC Estimates

Size	XOR gates	AND gates	Delay
4-bit[4]	15	16	$T_A + 3T_X$
8-bit[6]	72	64	$T_A + 6T_X$
16-bit[6]	285	256	$T_A + 7T_X$
32-bit[6]	1,085	1,024	$T_A + 9T_X$
64-bit[6]	4,160	4,096	$T_A + 9T_X$
128-bit[6]	16,637	16,384	$T_A + 10T_X$

Even when compared to the ideal Mastrovito multiplier, the auto-generated multiplier remains very competitive. The gap between the two never exceeds one T_X (XOR delay) for any operand size. In both cases where the auto-generated multiplier lags behind the ideal Mastrovito multiplier, $p(x)$ is of the form $x^n + x^4 + x^3 + x + 1$. For this type of pentanomial, the synthesis tool seems unable to completely optimize partial product generation. This may in fact be due to the limitation of auto-generation or it may be simply a weakness in the synthesis tool. More detailed analysis of the resulting networks will need to be performed before results like [6] can be obtained for these polynomials.

VII. FPGA Implementation

Automatic generation of multipliers in $GF(2^n)$ is particularly interesting for FPGAs, given the large number of polynomials that are available in the standard. For more security, several designs could be generated and replaced periodically. The flexibility of the FPGA, combined with the auto-generation capabilities, would allow for rapid changes and make periodic design changes quick and cost effective.

Given the resources available within an FPGA, pipelining the multiplier becomes a very effective way to improve performance without much of an impact on size. By pipelining the auto-generated multiplier, the performance hit, caused by the nature of FPGA implementations, could be significantly reduced. This change will greatly improve the throughput of the auto-generated multiplier.

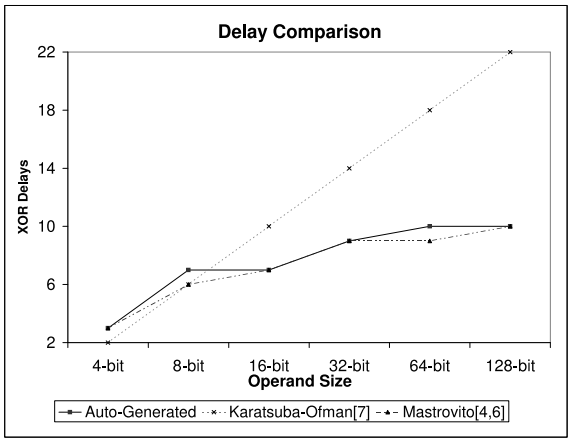

Fig. 2. Our auto-generated multiplier out performs the Karatsuba multiplier for larger values of n and remains very close in performance to the ideal Mastrovito multiplier.

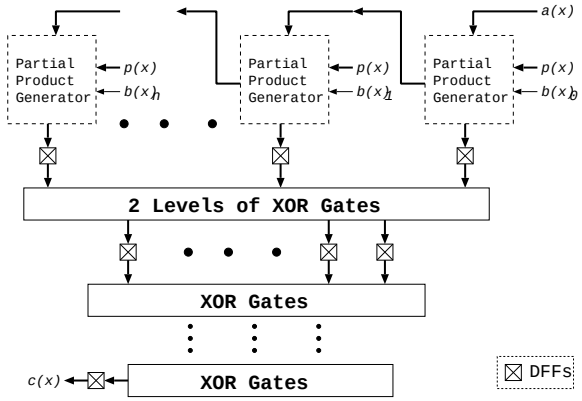

Fig. 3. Block diagram of pipelined FPGA implementation

The original recursive design was modified to include input and output registers as well as registers to hold each partial product. Flip-flops were also added to the recursive XOR trees. The flip-flops were inserted between every other level of the tree structure as shown in 3. The recursive VHDL allowed for easy integration of these flip-flops into the XOR trees. For details see Section VIII. The results of the FPGA synthesis for the pipelined implementation are shown in Table IV.

For the target of our FPGA synthesis we again used Leonardo. We chose Xilinix VIRTEX as our target family. The specific device was varied depending on the space requirements of the multiplier being generated.

To study the performance of the pipelined recursive multiplier, a non-pipelined version was also synthesized to the same FPGAs. The synthesis results for this multiplier are shown in Tables V.

As expected, the throughput of the auto-generated multiplier was greatly increased by pipelining. See Fig. 4 for a comparison of throughput for the two designs. While the latency of a single calculation would be less in the non-pipelined design, the ability of the pipelined multiplier to complete a multiplication every clock cycles makes it faster when considering a large number of multiplications performed back to back.

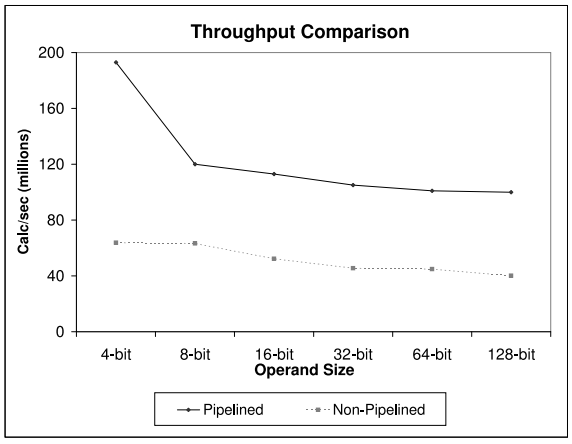

Fig. 4. Pipelined performance does not include cold start

Table V
Non-Pipelined Multiplier FPGA Results

Size	LUTs	CLB Slices	Delay (ns)
4-bit	13	7	15.67
8-bit	84	42	15.81
16-bit	324	162	19.10
32-bit	1,365	683	21.99
64-bit	5,435	2,718	22.30
128-bit	21,760	10,880	24.95

Table IV
Pipelined Recursive Multiplier FPGA Results

Size	LUTs	CLB Slices	Dff	Clk (MHz)
4-bit	21	14	28	193
8-bit	86	44	88	120
16-bit	358	184	368	113
32-bit	1,363	682	1,259	105
64-bit	5,592	2,819	5,638	101
128-bit*	19,101	—	18,110	100

*Unoptimized, only an initial pass could be made.

Currently, the main limiting factor in performance of the pipelined recursive design is the partial product generation stage. To allow the synthesis tool to freely optimize partial product generation, D-flip-flops cannot be inserted into this stage prior to synthesis. The key to improving performance at larger operand sizes will be to modify the partial product generation to allow for automated pipelining without impairing the synthesis tool's ability to optimize this stage based on the polynomial.

VIII. Pipelining the Recursive VHDL Trees

To make the multiplier more effective in an FPGA environment, the design was pipelined to increase throughput and clock speed. To pipeline the recursive tree structures, additional generic parameters are passed between levels. The generic values indicate whether or not the output of that level should be latched. One value sets the number of recursive

levels in between latches. The second determines which levels will actually be latched. This inherit flexibility allows for the pipelined recursive tree to be optimized for clock frequency and target device.

The code for tree1 was modified in the following manor:

```
BEGIN // beginning of architecture body
...
  compound_tree: IF d > 1 GENERATE
   IF L = 0 THEN
    the_gate1: xor_n GENERIC MAP(n => n)
     PORT MAP(a => xor_input,
      z => temp_out);
    tree_ff: PROCESS(reset, clk, temp_out)
    BEGIN
     IF reset = 1 THEN
      output <= '0';
     ELSIF clk'EVENT AND clk='1' THEN
      output <= temp_out;
     END IF;
    END PROCESS tree_ff;
    subtree_array1: FOR i IN 0 TO n-1
    GENERATE
     the_subtree1: tree1
     GENERIC MAP(d => d-1, n => n,
      L => c, c => c)
     PORT MAP
      (input => input(i*n**(d-1) TO
       (i+1)*n**(d-1)-1),
      output => xor_input(i),
      clk => clk, reset => reset);
    END GENERATE subtree_array1;
   ELSE
    the_gate2: xor_n GENERIC MAP(n => n)
     PORT MAP(a => xor_input,
      z => output);
    subtree_array2: FOR i IN 0 TO n-1
    GENERATE
     the_subtree2: tree1
     GENERIC MAP(d => d-1, n => n,
      L => L-1, c => c)
     PORT MAP
      (input => input(i*n**(d-1) TO
       (i+1)*n**(d-1)-1),
      output => xor_input(i),
      clk => clk, reset => reset);
    END GENERATE subtree_array2;
   END IF;
  END GENERATE compound_tree;
END; // end of architecture body
```

L is used to determine if a particular level within the tree should be latched. c is the number of levels between latches minus one. When $L = 0$, a D-flip-flop is created to hold the output of the XOR gate. Each sub-tree created for the levels above will have L mapped to c. For $L > 0$, the level is created almost identically to the original recursive code, except for the additional generic and port mappings. In this case, the sub trees are created with L mapped to $L - 1$.

This implementation not only allows for control over the number of pipeline stages within the tree, but also where those stages are inserted. Changing the value of L passed to the top level of the recursive tree will move the latches up or down within the tree. This would allow the final result to be latched within the tree or allow the last level(s) to be used and then latched externally of tree. c can be adjusted to improve clock frequency or overall latency depending on the implementation. Larger values of c will mean fewer pipeline stages with a lower clock frequency. Smaller values would create more stages and allow for a higher clock frequency.

IX. Conclusion

As the development environment continues to call for shorter and shorter cycles, auto-generation will become more and more critical to successful product development. Moreover, the ability to generate a dedicated solution only based on general parameters is of great interest to a system that has FPGAs. The results obtained with the synthesis of this flexible VHDL description for both AMI-CMOS and FPGAs are comparable to those created by traditional and optimized design methods. The results show that automatic generation of polynomial basis multiplier in $GF(2^n)$ is a quick, flexible and efficient alternative to traditional design.

It was when attempting to pipeline the auto-generated multiplier, that the real advantage of the recursive VHDL became apparent. The ability of recursion to more accurately model tree structures makes it clearly superior to cruder descriptions. When used effectively, recursion can greatly improve the development of tree structures in VHDL.

The test results we acquired in terms of delay versus area are promising, and we hope that this design approach for auto-generating multipliers in $GF(2^n)$ can still be improved to lead to better optimized designs and also to generate new ideas for other designs.

X. Acknowledgements

The authors would like to acknowledge the partial support of NSF from the CAREER Grant CCR-0093434 - "Computer Arithmetic Algorithms and Scalable Hardware Designs for Cryptographic Applications".

References

[1] S. B. Wicker and V. K. Bhargava, *Reed-Solomon Codes and Their Applications*. IEEE Press, New York.

[2] E. D. Mastrovito, "VLSI designs for multiplication over finite fields GF(2^m)," in *Proceedings of the 6th International Conference, on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes.* Springer-Verlag, 1989, pp. 297–309.

[3] IEEE P1363, "Standard specifications for public key cryptography," *Draft 13*, November 1999, http://grouper.ieee.org/groups/1363/.

[4] B. Sunar and Ç. K. Koç, "Mastrovito multiplier for all trinomials," *IEEE Trans. Comput.*, vol. 48, no. 5, pp. 522–527, 1999.

[5] A. Halbutoğullari and Çetin K. Koç, "Mastrovito multiplier for general irreducible polynomials," *IEEE Trans. Comput.*, vol. 49, no. 5, pp. 503–518, 2000.

[6] A. Reyhani-Masoleh and M. A. Hasan, "On low complexity bit parallel polynomial basis multipliers," in *Cryptographic Hardware and Embedded Systems*, ser. Lecture Notes in Computer Science, No. 2779, C. D. Walter, Ç. K. Koç, and C. Paar, Eds. Springer, Berlin, Germany, 2003, pp. 189–202.

[7] F. R. Henriquez, “Research problem: Fully parallel multipliers for GF(2^m),” CINVESTAV-IPN #2508, http://delta.cs.cinvestav.mx/ adiaz/RecComp2003/Rproblems.pdf.

[8] P. J. Ashenden, “Recursive and repetitive hardware models in VHDL,” technical Report TR 160/12/93/ECE, Department of Electrical and Computer Engineering, University of Cincinnati, also published as Technical Report 93-19, Department of Computer Science, The University of Adelaide, Australia.