

# Synthesis of Small and Fast Finite Field Multipliers for Field Programmable Gate Arrays \*

Gregory C Ahlquist<sup>†</sup>  
School of Systems and Logistics  
Air Force Institute of Technology, Wright-Patterson AFB, OH 45433  
Gregory.Ahlquist@afit.edu

Brent E. Nelson and Michael D. Rice  
Department of Electrical and Computer Engineering  
Brigham Young University, Provo, UT 84602

## 1 Abstract

The efficient implementation of finite field multipliers on Field Programmable Gate Arrays (FPGAs) is an enabling technology for Software Defined Radios, Dynamic Cryptography, and other applications that depend on fast and flexible finite field circuits. The associated finite field multiplier-FPGA design problem possesses cutting-edge characteristics that combine to define a unique and challenging synthesis problem. These characteristics include mapping to Look-Up-Table (LUT) based FPGA architectures, multiple-input, multiple-output functions, Exclusive-or Sum-Of-Products (ESOP) equation representations, and performance-enhancing measures such as pipelining. We present an efficient algorithm that exploits these characteristics to significantly reduce circuit size and improve circuit performance. The resulting designs are fifty to sixty percent smaller and from ten to twenty percent faster than circuits synthesized by competing methods.

## 2 Introduction

Next generation technologies such as Software Defined Radios (SDR) [2, 4] and dynamic cryptography [2] require circuits exhibiting the speed of hardware coupled with the flexibility of software. As such, these technologies can use Field Programmable Gate Arrays (FPGAs) to realize the fast and flexible applications required. One sub-class of these applications includes Reed-Solomon error control coding, encryption, and other capabilities that depend on

finite field multiplication to work. Understanding how to efficiently implement finite field multiplication circuits on FPGAs, therefore, directly supports the implementation of finite field circuits on FPGAs and, subsequently, the realization of promising next generation dynamic systems (Figure 1).

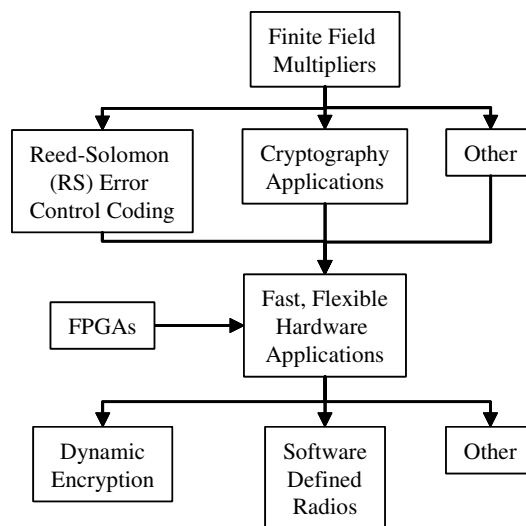


Figure 1: Relationships Between Finite Field Multipliers, FPGAs, and the Applications They Support

When implementing finite field circuits in general, the size and speed of finite field multipliers often dictates the size and speed of the overall circuit. This effect is magnified in FPGAs due to the relatively coarse size of the fundamental circuit elements — the logic block. Reducing the size and boosting the performance of the finite field multiplier circuit, therefore, has a direct, positive impact on

\*This research is sponsored in part by the United States Air Force and has been approved for public release; Distribution A, Distribution Unlimited. The views expressed in this article are solely those of the authors. They should not be construed as official policy of the United States Air Force, the Department of Defense, or the United States Government

<sup>†</sup>This primary author accomplished the majority of this research while at Brigham Young University.

the number of finite field applications a given FPGA can support. For this reason, efficient finite field multiplier designs for FPGAs, and any related Computer Aided Design (CAD) synthesis tools, become valuable commodities.

Equations 1 through 3 below represent the governing Boolean equations for a specific instance of a finite field multiplier. The set of equations possesses certain properties to include multiple-inputs and multiple-outputs as well as equations completely expressed as Exclusive-or Sum-Of-Products (ESOPs). As such, we refer to equations 1 through 3 collectively as a *ESOP switching function* and to the problem of synthesizing these equations to an FPGA as an *ESOP switching function synthesis problem*.

$$c_0 = a_0b_0 \oplus a_1b_2 \oplus a_2b_1 \quad (1)$$

$$c_1 = a_0b_1 \oplus a_1b_0 \oplus a_1b_2 \oplus a_2b_1 \oplus a_2b_2 \quad (2)$$

$$c_2 = a_0b_2 \oplus a_1b_1 \oplus a_2b_0 \oplus a_2b_2 \quad (3)$$

Traditional synthesis methods are fully capable of synthesizing a finite field multiplier circuit for FPGAs from the ESOP switching function expressed above. We have observed [1], however, that smaller and faster circuits can be realized by exploiting the specific characteristics of the ESOP switching function synthesis problem especially in the case of pipelined circuits. For example, traditional logic synthesis methods coupled with retiming can produce the FPGA implementation of our example ESOP switching function as shown in Figure 2. This implementation is fully pipelined and requires ten total 4-input Logic Blocks (Look-Up-Table – register pairs) to implement.

In comparison, we have derived an FPGA implementation for the same ESOP switching function as shown in Figure 3. This circuit is also fully pipelined but requires twenty percent fewer logic blocks and a correspondingly reduced amount of interconnect to implement.

Based on our observations, we have codified our approach and developed an automated synthesis method that produces highly efficient finite field multiplier circuits for 4-input Look-Up-Table FPGAs. Our results are noteworthy.

- On average, our method synthesizes pipelined designs that are 40 percent smaller than circuits synthesized by competing methods.
- In some cases, our method synthesizes pipelined designs that are up to 65 percent smaller than circuits synthesized by competing methods.
- In addition, circuits synthesized by our method possess fewer and shorter interconnects which result in 10 to 20 percent faster clock cycle speeds than circuits synthesized by competing methods.

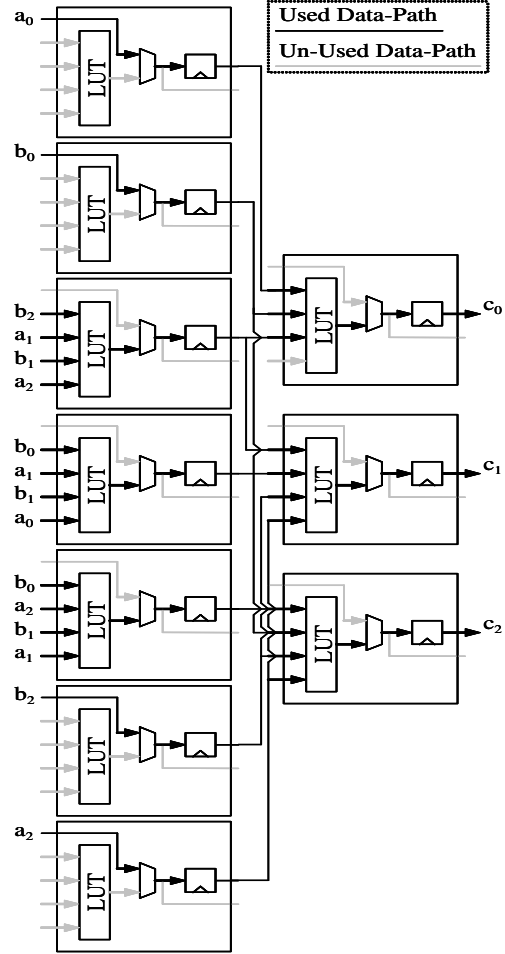


Figure 2: Implementation of a Finite Field Multiplier on an FPGA as Synthesized by Traditional Methods

At the heart of this synthesis method is a highly counter-intuitive algorithm that essentially *decreases* the total number of required logic blocks by *increasing* the number of terms in the ESOP switching function. Terms are added to the ESOP switching function through the use of the exclusive-or (XOR) identity  $b \oplus b = 0$  which allows us to duplicate logic in the system of equations without affecting the overall global result. We find that if we add terms correctly we can exploit the ability of a logic block to produce any logical combination of its inputs in a fixed circuit area and derive FPGA circuit mappings that require fewer logic blocks. In the following sections, we present our automated synthesis methods in detail.

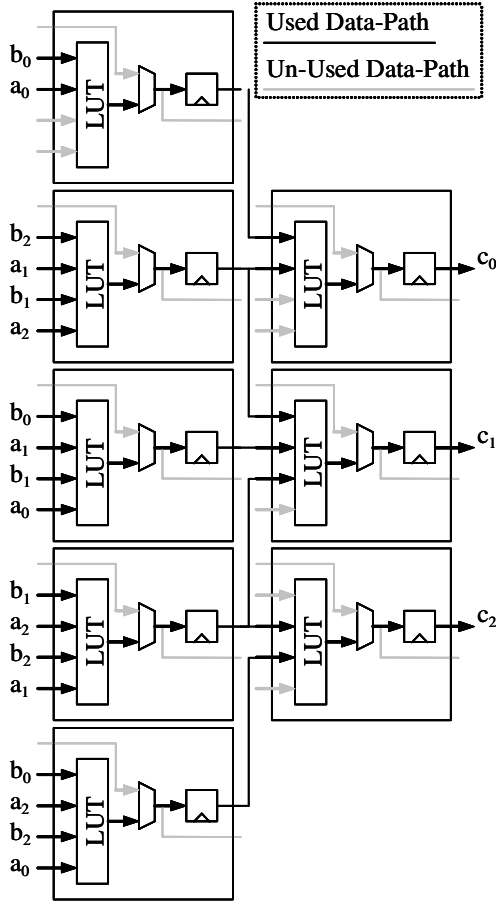


Figure 3: Alternative Implementation of the Same Finite Field Multiplier as Produced by Hand-Mapping

### 3 Problem Formulation and Preliminaries

#### 3.1 Finite Fields

The finite field  $GF(2^m)$ <sup>1</sup> consists of the finite set of all binary polynomials of degree  $m - 1$  (Example:  $0, 1, x,$  and  $x + 1$  for  $m = 2$ ) and is *closed* under the addition and multiplication operations. The closed property means that the addition or multiplication of any two elements of the field results in another element of the field. Addition, therefore, is defined as the standard polynomial addition of two elements except the coefficient arithmetic is performed modulo-2. The much more complex multiplication operation is defined as the modulo- $g(x)$  reduction of the polynomial product as shown in Equation 4. The reduction polynomial  $g(x)$  is called the *degree- $m$  generator polynomial* for the field [5].

<sup>1</sup>The value  $m$  is used on numerous occasions in the following discussions. In all instances, it relates to the size of the finite field  $GF(2^m)$ .

$$c(x) = a(x) * b(x) \text{ mod } g(x) \quad (4)$$

Expanding equation 4 produces a set of Boolean relations that represent the related finite field multiplier. For example, evaluating equation 4 with  $g(x) = x^3 + x + 1$  results in the ESOP switching function expressed in equations 1 through 3. The governing Boolean equations for any finite field multiplier can be derived in this manner. As the ESOP switching function generated through equation 4 is wholly dependent on the generator polynomial  $g(x)$ , we can refer to it as  $S_{g(x)}$  in the general case and as  $S_{x^3+x+1}$  or  $S_{1011}$  in the specific case. In the second reference, we employ a shorthand notation derived by replacing the full binary polynomial with a listing of its binary coefficients.

#### 3.2 FPGA Architecture

Although variations abound, the most common FPGA architectures can be modeled as a two dimensional array of Logic Blocks (LBs) surrounded by interconnect as shown in Figure 4.

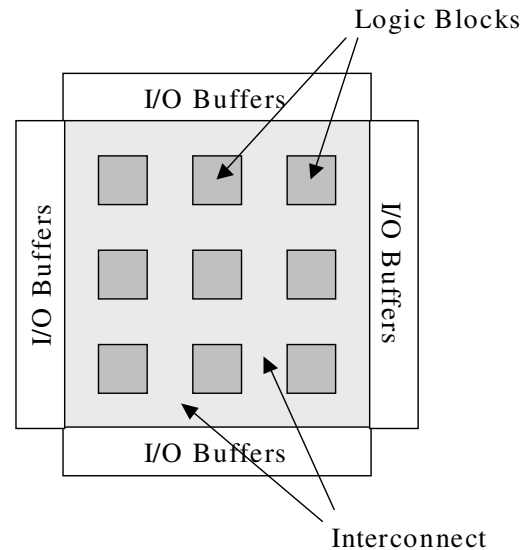


Figure 4: Simplified Diagram of an Array-Based FPGA

The logic blocks are generally 4-input, 1-output Look-Up-Tables (4-LUTs) paired with a flip-flop as shown in Figure 5. Each 4-LUT implements some Boolean function of its 4 inputs. Combinatorial circuits are realized by chaining LUTs together into larger and larger designs. Sequential or pipelined circuits are realized through the use of the flip-flops.

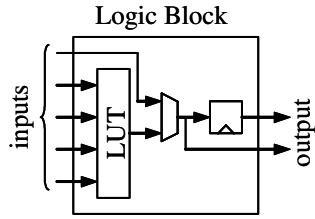


Figure 5: A Logic Block As A LUT/Flip-Flop Pair

### 3.3 Definitions and Terms

The following definitions and terms, which are largely attributable to Murgai[6], will be useful in our discussions.

In our work, we deal exclusively with multiple-output functions. A useful way to represent a multiple-output function is with a Boolean network.

**Definition 1** A Boolean or Logic Network  $n$  is a directed acyclic graph (DAG), with primary inputs  $PI(n)$ , primary outputs  $PO(n)$ , and internal (intermediate) nodes  $IN(n)$ . Primary inputs have no arcs coming into them, and primary outputs have no arcs going out of them. Associated with each internal node  $i$  of the network is a variable  $y_i$ , and a representation of a logic function  $f_i$ . There is a directed arc from node  $i$  to node  $j$  in the network if  $j$  uses  $y_i$  or  $y_i'$  explicitly in the representation of  $f_j$ . In that case,  $i$  is called a fanin of  $j$ , and  $j$  a fanout of  $i$ . The set of fanins of a node  $i$  is denoted as  $FI(i)$  and the set of fanouts as  $FO(i)$ . A node is a multi-fanout or multiple-fanout node if it has more than one fanout.

Figure 6 is an example of a Boolean network. It is the Boolean network for switching function  $S_{1011}$  as expressed in equations 1 through 3.

**Definition 2** For a given block of logic, a function  $f$  is feasible if it can be implemented by one logic block. A logic network  $n$  is feasible if the function at each internal node is feasible. In our specific case, we say a function  $f$  is 4-feasible if it can be implemented by a 4-input LUT. We say a logic network is 4-feasible if the function at each internal node is 4-feasible.

These definitions and terms will help us define and present more complex concepts in the following sections.

## 4 The Design Method

In his book on logic synthesis, De Micheli [3] presents numerous heuristic algorithms for optimizing switching functions using logic networks. He proposes a general approach of first producing a logic network that represents the switching function to be synthesized and then applying

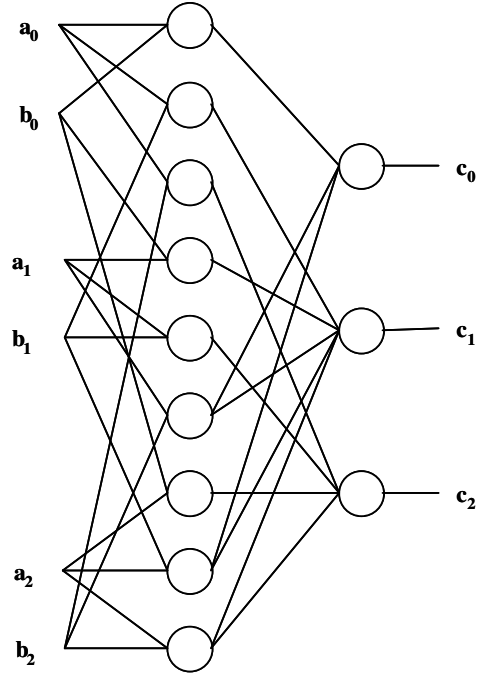


Figure 6: An Example Logic Network

a series of transformations to the logic network to produce a new network representing a circuit implementation optimized according to some cost function. Our design method follows De Micheli's outline and is shown in Figure 7.

The algorithm begins by constructing the switching function  $S_{g(x)}$  from the generator polynomial  $g(x)$  for a specific finite field. We then create an initial logic network representation of  $S_{g(x)}$  which we call the *Initial DAG*. We next apply two logic network transformations, Merge and XOR-Merge which work in sequence to reduce the node count of the logic network. Node count is our objective cost function we wish to optimize because the number of nodes in the logic network is roughly equivalent to the number of logic blocks required to implement the circuit in an FPGA. The Merge transformation reduces the number of nodes by merging two-input nodes into four-input nodes in a prescribed manner. We call the product of this transformation a *Merged DAG*. The XOR-Merge transformation then further reduces node count through the use of duplicate logic. We call the product of this transformation the *XOR-Merged DAG*. After these transformations, we evaluate the XOR-Merged DAG to determine if it is 4-feasible. If it is, we exit the algorithm and the XOR-Merged DAG represents our final circuit design. If the XOR-Merged DAG is not 4-feasible, we apply another transformation called Decompose which splits non-4-feasible nodes in the network into 4-feasible nodes. The DAG resulting from

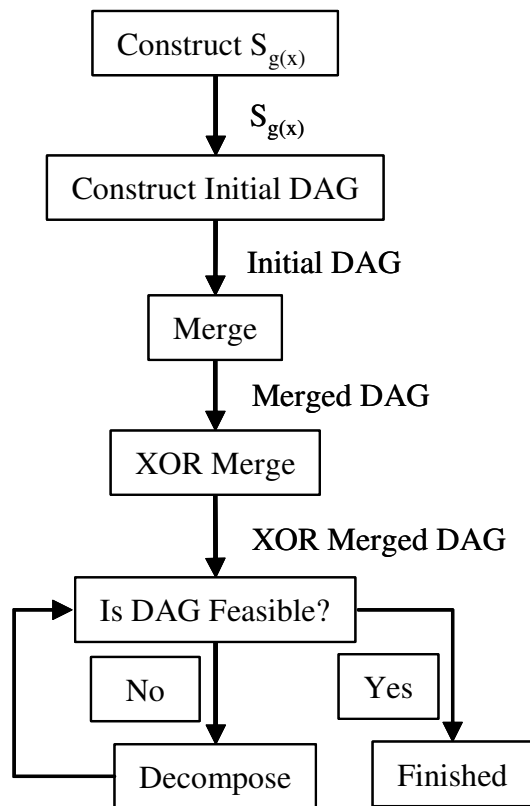


Figure 7: Design Method Process Flow

Decompose is again evaluated for feasibility and the algorithm continues to apply the Decompose transformation until a feasible DAG results. We discuss each of these transformations in more detail in the following sections.

#### 4.1 Preliminary Steps and The Initial DAG

The Initial DAG required by our design method is a logical representation of the switching function  $S_{g(x)}$  which is derived from a generator polynomial  $g(x)$  through the application of Equation 4. The Initial DAG can be viewed in two ways. First, it calculates the switching function  $S_{g(x)}$  where the internal nodes compute the product terms and the output nodes sum terms appropriately to realize the switching function outputs. The Initial DAG can also be viewed as a circuit diagram where every node represents a logic block and the edges into and out of nodes are interconnections between logic blocks. If we assume that all node outputs are latched, the Initial DAG also represents a pipelined implementation of the circuit. As an example, Figure 6 represents the initial DAG for  $S_{1011}$ .

The second view, or the circuit view, of the logic network is the view our synthesize algorithm uses. As each node in the logic network thus represents a logic block, it

becomes clear that our algorithm's main goal is to minimize the number of nodes in the logic network. For performance reasons, however, we also want to maintain the pipelined architecture exhibited by the Initial DAG. These are the challenges that drive the Merge and XOR-Merge transformations.

#### 4.2 The Merge Transformation

The Merge transformation exploits the fact that the product terms  $a_i b_j$  and  $a_j b_i$  always appear in the same equations in  $S_{g(x)}$  (something that is true for any multiplication function). The transformation therefore simply pairs and merges the nodes producing the  $a_i b_j - a_j b_i$  term pairs. The resulting nodes receive four inputs ( $a_i, b_j, a_j,$  and  $b_i$ ) and therefore naturally map to the 4-input LUTs of our target FPGA architecture. Also, the resulting Merge DAG will still correctly calculate  $S_{g(x)}$  and exhibit a pipelined architecture after the transformation. For example, applying the Merge transformation to the initial DAG of Figure 6 combines the nodes producing terms  $a_0 b_1$  and  $a_1 b_0$  into a new node  $p_{01}$  as shown in Figure 8. Node  $p_{01}$  has four inputs ( $a_0, b_1, a_1,$  and  $b_0$ ) and produces the function  $a_0 b_1 \oplus a_1 b_0$ . Other new nodes produced by applying the Merge transformation to our Initial DAG include  $p_{02}$  and  $p_{12}$  which produce functions  $a_0 b_2 \oplus a_2 b_0$ , and  $a_1 b_2 \oplus a_2 b_1$  respectively. The Merged DAG resulting from the application of our Merge transformation to the Initial DAG of Figure 6 is illustrated in Figure 9.

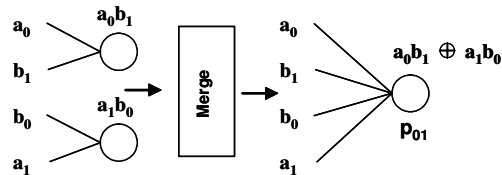


Figure 8: Merging Two Nodes to Create One

#### 4.3 The XOR-Merge Transformation

The XOR-Merge transformation, while still a merging operation, is far more complex than Merge. It has as its primary focus all the nodes in the Initial DAG that were unaffected by the Merge transformation. By definition, these nodes all share the property that they produce an  $a_k b_k$  term. The XOR-Merge Transformation seeks to pair and merge  $a_k b_k$  nodes in order to reduce the overall node count in the network. A simple example shows how this can be done.

**Example 1** Consider node  $a_2 b_2$  with input set  $(a_2, b_2)$ , output set  $(c_1, c_2)$  and function  $a_2 b_2$  in the Merged DAG

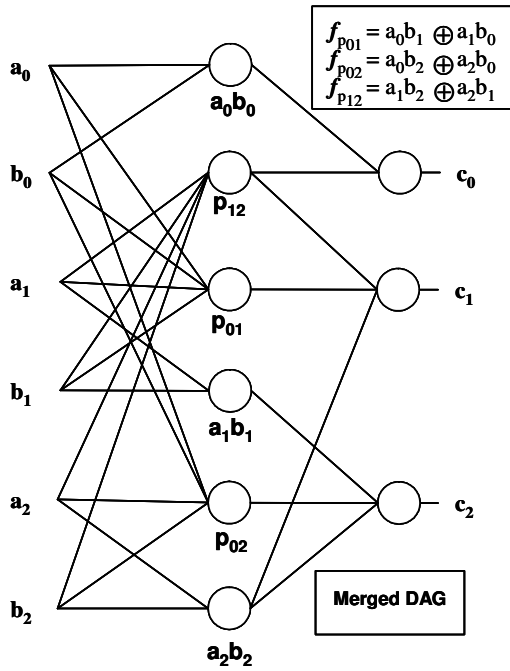


Figure 9: The Merged DAG

of Figure 9. With respect to this node, we make the following modifications to the Merge DAG as shown in Figure 10

1. Modify the function of the node  $p_{02}$  from  $a_0b_2 \oplus a_2b_0$  to  $a_0b_2 \oplus a_2b_0 \oplus a_2b_2$ . This change supplies a new  $a_2b_2$  term to the input of node  $c_2$ .
2. The new  $a_2b_2$  term from  $p_{02}$  allows us to remove the edge from  $a_2b_2$  to  $c_2$ . This is shown by the dark X on that edge in the lower-rightmost corner of Figure 10.
3. In like manner, we modify the function of  $p_{12}$  from  $a_1b_2 \oplus a_2b_1$  to  $a_1b_2 \oplus a_2b_1 \oplus a_2b_2$ . This supplies a new  $a_2b_2$  term to the input of node  $c_1$  and allows us to remove the edge from  $a_2b_2$  to  $c_1$ .
4. Unfortunately, this last action introduced a negative side effect — node  $c_0$  now receives an unneeded  $a_2b_2$  term from  $p_{12}$ . As this changes the switching function represented by the network, we must eliminate this error. To do this, we add an edge from  $a_2b_2$  to  $c_0$  to supply a second  $a_2b_2$  term to the input of node  $c_0$ . By virtue of the XOR identity  $b \oplus b = 0$ , these two terms cancel each other out and the error is eliminated. The gray line from  $a_2b_2$  to  $c_0$  in Figure 10 shows this new edge.
5. The nodes  $a_2b_2$  and  $a_0b_0$  in our Merged DAG now have the same output sets ( $c_0$ ). As such, they can be

paired and merged. Thus, we have successfully removed one node from the Merged DAG as shown in Figure 11.

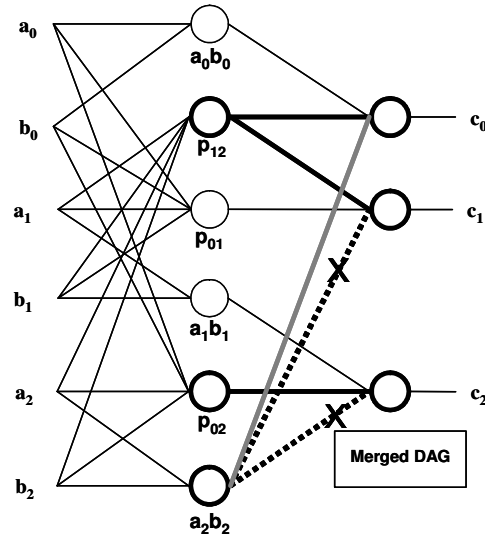


Figure 10: Actions of XOR-Merge on a Merged DAG

To help explain our example, it's important to remember that our logic network dually represents an ESOP switching function and a logic circuit featuring 4-LUT based logic blocks. Two key properties of 4-LUTs and ESOP switching functions combine to make this example work. First, a 4-LUT can produce any logical combination of its inputs in a given circuit area. This allows us to modify the function of any node in the logic network to produce additional product terms without increasing overall area. For example, the node  $p_{12}$  realizing function  $a_1b_2 \oplus a_2b_1$  can also realize function  $a_1b_2 \oplus a_2b_1 \oplus a_2b_2$  with no increase in area. Second, the XOR identity  $b \oplus b = 0$  makes it possible to repair side effects caused by modifying node functions to produce additional terms. This allows us to always ensure our logic network appropriately calculates the original switching function. These two ideas, when combined, present new opportunities for LB mapping that result in smaller circuits.

At this point, it's insightful to derive and evaluate the switching function represented by the DAG in Figure 11. As shown in Equations 5 through 7 our changes to the logic network introduced duplicate logic — specifically the  $a_2b_2 \oplus a_2b_2$  sub-expression in Equation 7. Thus, we actually increased the number of terms realized in order to decrease the number of nodes in the related DAG. This counter-intuitive result is similar to those achieved through duplicate logic FPGA logic synthesis strategies suggested

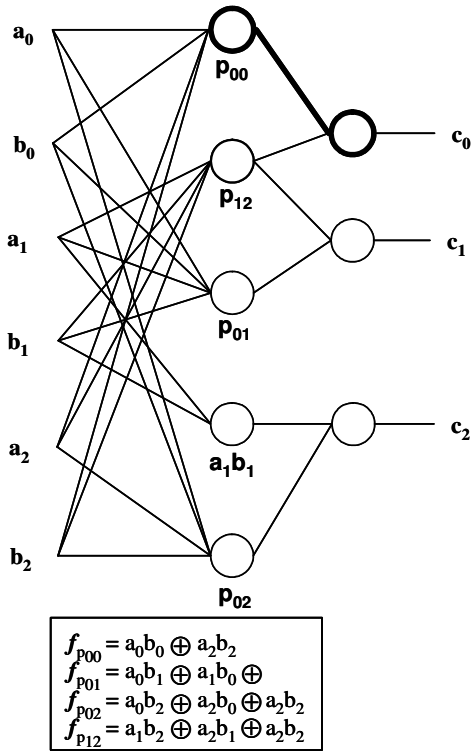


Figure 11: Example After XOR-Merge

by Murgai[6] and others. In essence, the duplicate logic allows us to derive a more efficient mapping.

$$c_2 = a_0b_2 \oplus a_2b_0 \oplus a_2b_2 \oplus a_1b_1 \quad (5)$$

$$c_1 = a_0b_1 \oplus a_1b_0 \oplus a_1b_2 \oplus a_2b_1 \oplus a_2b_2 \quad (6)$$

$$c_0 = a_0b_0 \oplus a_2b_2 \oplus a_1b_2 \oplus a_2b_1 \oplus a_2b_2 \quad (7)$$

◇

Example 1 suggests a general strategy for our XOR-Merge Transformation. We want to evaluate and find all instances where duplicating logic allows us to pair and merge  $a_k b_k$  nodes. Like our example, we have found it easiest to illustrate the XOR-Merge transformation in terms of manipulations to our logic network. From this perspective, two important concepts are key to presenting the transformation in more detail.

**Definition 3** Within the Merged DAG there will be several nodes that can produce a given  $a_k b_k$  term. A reduction relative to term  $a_k b_k$  consists of a subset of these nodes. The subset may be empty which corresponds to the unmodified Merged DAG. We express a reduction in terms of the Merged DAG nodes modified. For example, the reduction

with respect to term  $a_2 b_2$  illustrated in Example 1 is  $\{p_{02}, p_{12}\}$ .

**Definition 4** A singleton relative to term  $a_k b_k$  is a reduction relative to  $a_k b_k$  that reduces the output set of the  $a_k b_k$  node to one item. For example, the reduction  $(p_{02}, p_{12})$  illustrated in Example 1 results in a singleton relative to node  $a_2 b_2$ .

Relative to these definitions, there are three main phases to the XOR-Merge transformation. First, we evaluate all possible reductions in a search for all singletons. These we record in a table. Second, we use the table of singletons to determine  $a_k b_k$  node pairs which can be merged. Finally, we use these pairings to guide the application of reductions to the unmodified Merged DAG to produce an XOR-Merged DAG. The combined application of these reductions is the XOR-Merge transformation. We present each of these phases in turn.

**Building the Singleton Table** Figure 12 details our algorithm for evaluating all reductions and building a table of singletons.

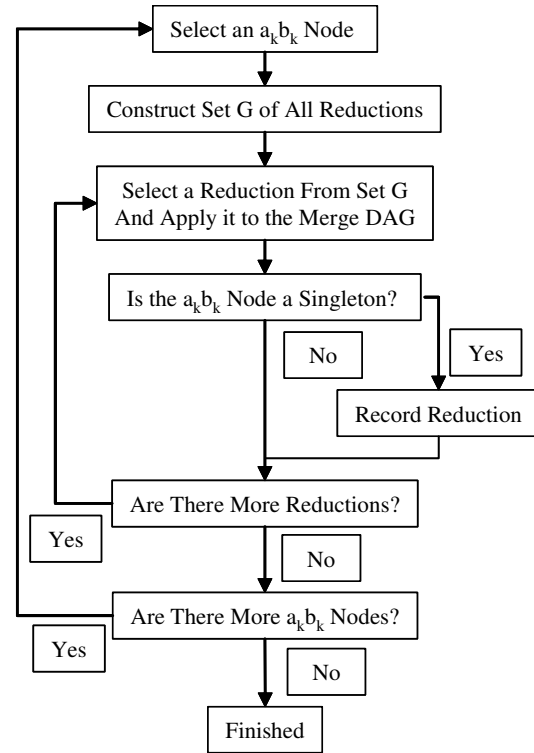


Figure 12: Algorithm for Building the Singleton Table

We begin by selecting an  $a_k b_k$  node from the Merged DAG and constructing a set  $G$  of all possible reductions

relative to  $a_k b_k$ . Set  $G$  is the power set of all previously merged nodes in the Merged DAG that can produce the term  $a_k b_k$ . For example, the set  $G$  of reductions relative to node  $a_2 b_2$  is the power set of  $\{p_{02}, p_{12}\} = \{\{\}, \{p_{02}\}, \{p_{12}\}, \{p_{02}, p_{12}\}\}$ . We apply each reduction in  $G$  to the unmodified Merge DAG and determine its effect on the output set of the related  $a_k b_k$  node. If a singleton results, we record the reduction in the singleton table. The singleton table is indexed using the node  $a_k b_k$  as the column index and the output node that it drives in the singleton condition as the row index. When all reductions for all the  $a_k b_k$  nodes in the Merged DAG are evaluated, the algorithm stops.

-	$a_0 b_0$	$a_1 b_1$	$a_2 b_2$
$c_0$	{}		$\{p_{02}, p_{12}\}$
$c_1$			$\{p_{02}\}$
$c_2$		{}	

Table 1: A Singleton Table

The completed singleton table (Table 1) shows how the output sets of each  $a_k b_k$  node can be reduced to a single entry and what that entry is. With this information, we can determine how  $a_k b_k$  nodes can be paired and merged after reduction. In some cases, the pairing is trivial. For example, the only pairing from Table 1 that reduces the number of nodes in the Merged DAG is to (a) perform the modifications to nodes  $p_{02}$  and  $p_{12}$  as indicated by the entries for  $a_2 b_2/c_0$ , (b) combine nodes  $a_0 b_0$  and  $a_2 b_2$  into  $p_{00}$ , and (c) feed its output to  $c_0$ . This result in the logic network shown in Figure 11. In most cases, however, deciding how to pair reduced  $a_k b_k$  nodes requires a simple test.

**Finding All Covers of the Singleton Table** In general, numerous combinations of pairings of singleton  $a_k b_k$  nodes may be evident in the singleton table. We need a complete algorithm to evaluate all such combinations to determine a minimum solution. Fortunately, the needed algorithm is relatively simple and similar to standard and well-understood table covering algorithms. We say, therefore, that our algorithm computes and evaluates all the *covers* of the singleton table. A cover is a set of (row, column) indices into the singleton table. For example, cover  $(0,0)(2,1)(0,2)$  represents one possible cover. Our algorithm searches the singleton table for all *valid* covers as defined by the following criteria.

1. Every column of the singleton table must have exactly one entry covered.
2. No blank entry in the table may be part of a cover.

3. *Case a:* If the number of switching function outputs is even: any row must have an even (or zero) number of entries covered.

*Case b:* If the number of switching function outputs is odd: exactly one row may have an odd number of entries covered. Remaining rows must have an even number (or zero) entries covered.

This emphasis on even row entries facilitates the pairing of nodes. As an example, Table 1 possesses two potential covers,  $\{(0,0)(2,1)(0,2)\}$  and  $\{(0,0)(2,1)(1,2)\}$ , that meet both criteria (1) and (2). Cover  $\{(0,0)(2,1)(1,2)\}$ , however, fails requirement (3b)(more than one row is covered an odd number of times) and is discarded. Cover  $\{(0,0)(2,1)(0,2)\}$  meets all three requirements and is a valid cover. It leads to the solution shown in Figure 11.

The singleton table for our example produces only one valid cover - Cover  $\{(0,0)(2,1)(0,2)\}$ . In general, a given singleton table will produce a small set of valid covers where each defines a unique XOR-Merge transformation. To determine the best cover among the set, we apply Equation 8 to produce an estimate of the number of LBs required to complete the circuit mapping given the application of a given cover. We then simply select the cover producing the lowest estimate. In Equation 8,  $m$  is the number of switching function outputs and  $|\sigma(c_i)|$  is the number of inputs to node  $c_i$  after the cover is applied. Equation 8 essentially estimates the number of feasible nodes we must add to the logic network to make any unfeasible output nodes feasible while retaining a pipelined architecture. The estimate is derived by dividing the number of node inputs for each output node ( $-\sigma(c_i)-$ ) by the number of LUT inputs (4) and summing across the  $m$  output nodes. The ceiling function accounts for cases where the number of inputs for a given output node does not evenly divide into 4 — a situation that necessitates adding another node to the network.

$$N = \sum_{i=0}^{m-1} \left\lceil \frac{|\sigma(c_i)|}{4} \right\rceil \quad (8)$$

**Applying a Cover** Once a cover is selected, we apply it to our Merged DAG to produce an XOR-Merged DAG. This is done by applying each reduction recorded in the singleton table as indicated by the cover. For example, the cover  $\{(0,0)(2,1)(0,2)\}$  calls for the application of the reductions recorded in the indicated table locations to the Merged DAG. The first two entries map to empty set reductions ( $\{\}$ ) so no modifications are required. The last entry, however, modifies nodes  $p_{02}$  and  $p_{12}$  to produce term  $a_2 b_2$  and reduces the output set of node  $a_2 b_2$  to  $c_0$ . These changes result in the XOR-Merged DAG of Figure 11.

Following our algorithm always ensures that the XOR-Merged DAG resulting from the XOR-Merge transformation accurately calculates the desired switching function  $S_{g(x)}$  and describes a pipelined architecture. For smaller multipliers, the XOR-Merge DAG may even be 4-feasible. If so, our XOR-Merged DAG represents the FPGA design we seek. If not, however, we must transform the DAG yet again to convert the non-4-feasible nodes 4-feasible ones. This final transformation is the Decompose transformation.

#### 4.4 The Decompose Transformation

The Decompose transformation is the last transformation we apply to our logic network. Its goal is to convert a non-4-feasible network into a 4-feasible one. This may require several iterations of the transformation but the end result will be a DAG that represents a valid, 4-feasible, FPGA circuit implementation.

The operation of our synthesis algorithm to this point guarantees two facts affecting the Decompose transformation. First, the transformation will only be invoked on non-4-feasible logic networks. Thus, we are guaranteed that at least one node in the network is non-4-feasible. Second the only nodes that will be non-4-feasible are the output nodes  $c_i$  of the network. These facts allow us to focus the attention of the Decompose transformation exclusively on the output nodes of the logic network. Here it performs two key functions.

- The Decompose transformation must decompose any non-4-feasible output nodes in the logic network into a set of 4-feasible nodes and appropriately insert them into the logic network structure.
- The Decompose transformation must maintain the pipelined architecture of the logic network.

This second function may require the decompose transformation to insert delay nodes into the logic network to preserve the correct timing. These nodes correspond to LBs in the circuit representation that essentially function as delay registers.

Figure 13 overviews our Decompose transformation algorithm. The algorithm visits each output node  $c_i$  and determines if the node is 4-feasible. If it is not, the transformation replaces  $c_i$  with a set of 4-feasible nodes ( $s_0, s_1, \dots, s_r$ ) and connects the outputs of these nodes to a new output node. The inputs of the replaced node  $c_i$  are partitioned among the inputs of the inserted  $s$  nodes such that all  $s$  nodes are 4-feasible. One way to effect this partitioning uses a well known cube-packing algorithm overviewed in Murgai[6]. This partitioning is essentially a straight-forward bin-packing algorithm where the inputs are partitioned into groups of 4 or fewer and each group is

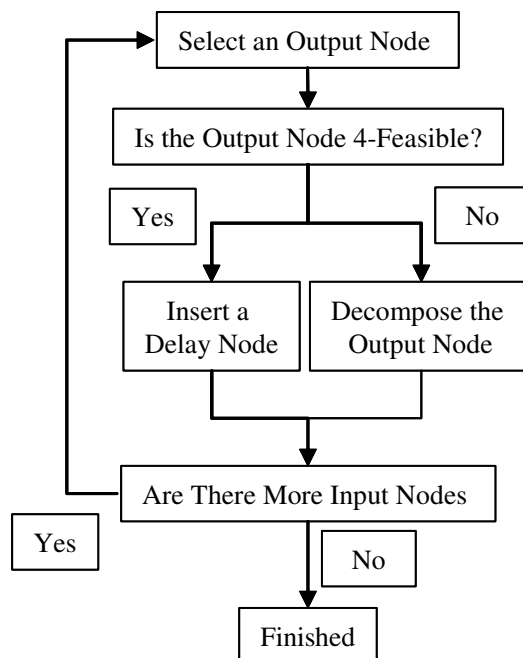


Figure 13: The Decompose Transformation Algorithm

assigned to a new  $s$  node. This concept is illustrated graphically in Figure 14.

In the case that the output node  $c_i$  is 4-feasible we must add a single node  $s$  to the network to appropriately delay the output and preserve the pipelined architecture. This new node replaces the old  $c_i$  node and drives a new output node. This scenario is shown graphically in Figure 15.

The decompose transformation is the one transformation we apply that actually increases the number of nodes in the logic network. This is unfortunate and runs against our goal of minimizing the number of nodes in the logic network but it is necessary in order to produce a feasible network corresponding to a valid pipelined FPGA circuit implementation. Nevertheless, we want to minimize the number of nodes we must add to the network which, in large measure, is a function of the partitioning algorithm we apply. The straight-forward cube-packing partitioning algorithm we described above is a workable approach but we have found it to be somewhat inefficient. Specifically, we can usually achieve better results in terms of fewer required nodes added through hand-mapping. We have observed, however, that we can improve the efficiency of the cube-packing partitioning algorithm in several ways. First we can employ common sub-expression extraction which views the logic network from a global perspective and seeks to extract sub-expressions common to two or more equations. Second, we can further extend the effi-

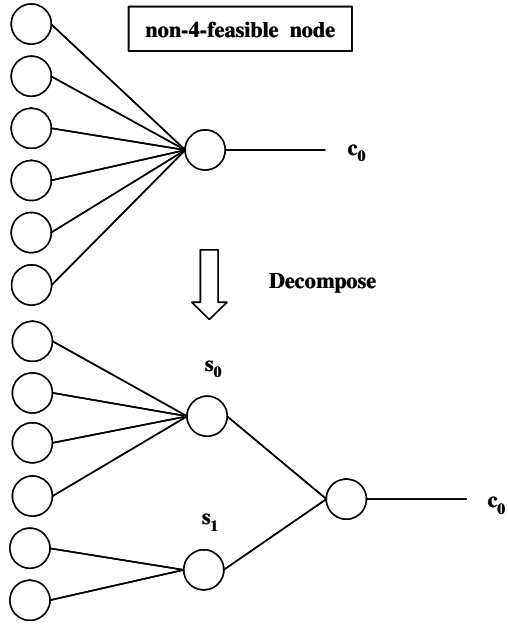


Figure 14: Decomposing a Non-4-Feasible Node

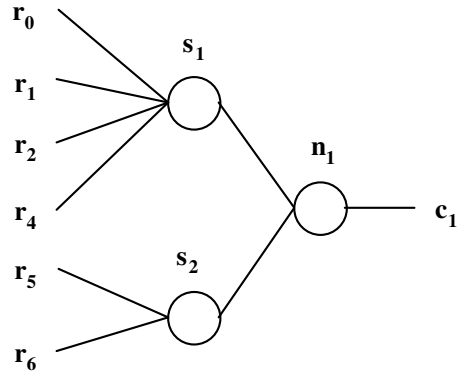
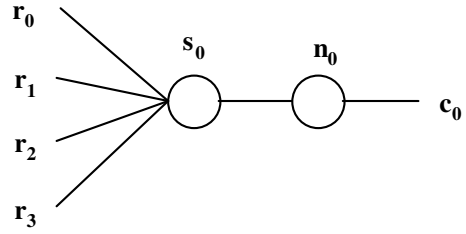


Figure 16: Cube-Packing Partitioning

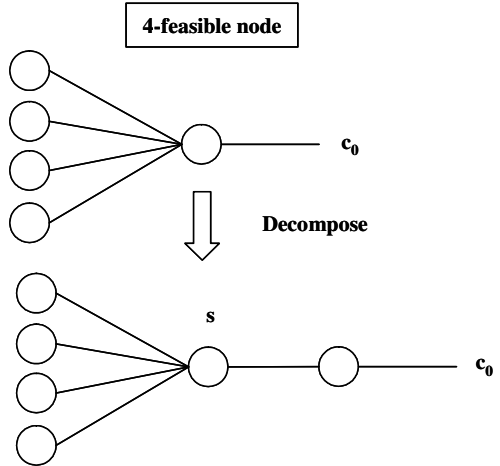


Figure 15: Decomposing a 4-Feasible Node

ciency of this common sub-expression extraction approach by employing duplicate logic and the XOR identity. The following example shows how.

**Example 2** Consider the following example ESOP switching function consisting of two equations.

$$c_0 = r_0 \oplus r_1 \oplus r_2 \oplus r_3 \quad (9)$$

$$c_1 = r_0 \oplus r_1 \oplus r_2 \oplus r_4 \oplus r_5 \oplus r_6 \quad (10)$$

If we apply straight-forward cube-packing partitioning to this switching function, we would require 5 total nodes to implement a pipelined mapping as shown in Figure 16. However, if we duplicate logic and add the expression  $r_3 \oplus r_3$  to Equation 10, we can create a node  $s_0$  usable by both functions as shown in Figure 17. This reduces the number of required nodes from 5 to 4.

◇

Using the ideas expressed in our example, we have developed an enhanced partitioning algorithm that globally examines all possible partitions of inputs to determine those that can be used to calculate more than one output. In doing so, we expressly look for opportunities to apply duplicate logic to expand the number of outputs a given

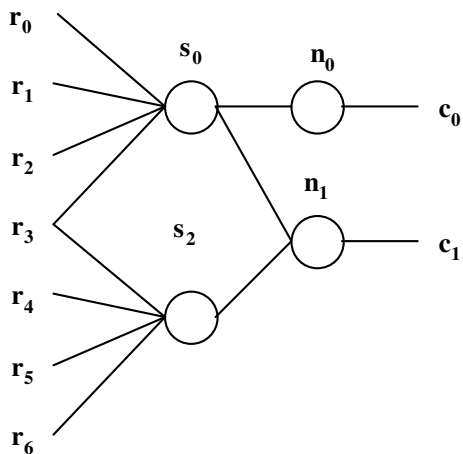


Figure 17: Extended Partitioning Using Duplicate Logic

partition can help calculate. To make our partitioning algorithm efficient, however, we consider only partitions where the majority of inputs in the partition drive more than one output. This reduces the set of possible partitions we must consider by excluding those that are essentially unique to the calculation of one output.

Applying our Decompose transformation using the partitioning algorithm described results in a new, pipelined DAG. This we must again evaluate to determine if it is 4-feasible. If so, our design is finished. If not, the Decompose transformation is repeatedly applied until a 4-feasible DAG results. The application of the Decompose transformation marks the end of our synthesis method. The 4-feasible DAG it produces represents the FPGA implementation we seek.

## 5 Results

To validate our approach, we have implemented our synthesis method in a complete suite of design, simulation, and testing software tools. These tools receive as an input the generator polynomial (expressed as a string of binary coefficients) for a given finite field multiplier and produce a pipelined finite field multiplier design for 4-input LUT based FPGA architectures. Appropriate area and timing information is extracted during the process and all multiplier designs are fully tested to ensure they correctly function as intended.

Table 2 contains area and timing data for ten pipelined multiplier designs produced by our synthesis method. These designs correspond to ten different finite fields of 256 elements ( $GF(2^8)$ ). The first two table columns record the generator polynomial used to generate the finite field and the total number of 4-input Logic Blocks (LBs)

required to implement the resulting finite field multiplier on an FPGA. The third column lists the number of logic blocks used solely as latches in the design which is an indicator of circuit balance. The last column is a measure of the circuit performance in terms of clock cycle time (in nano-seconds (ns)).

Generator Polynomial	Logic Blocks (LBs)	Latches Only	Clock Cycle Time (ns)
100011101	53	1	4.312
100101011	53	1	4.312
100101101	54	2	4.312
101100011	54	1	4.827
101100101	65	10	4.312
101101001	64	5	4.312
110001101	55	1	4.312
110101001	53	0	4.312
111100111	53	2	4.312
111110101	54	1	4.312

Table 2: Area and Timing Data for Pipelined Multiplier Designs

As we review the data for our pipelined designs, several trends emerge.

1. Area and (to a lesser extent) timing data varies with respect to the finite field involved. We find that, in general, the specific generator polynomial used has a direct affect on the area resources required to implement the multiplier circuit. Nevertheless, we have observed that this variation is far less with our method than with other synthesis methods we have studied. Our method produces designs with area measures ranging from 53 to 65 logic blocks required. This is in comparison to area measures ranging from 80 to 129 logic blocks required for other methods.
2. For our designs, the relative number of logic blocks used solely as latches is very small compared to the overall number of logic blocks in the design. This indicates that our designs are relatively balanced — meaning that the designs require approximately the same number of LUTs and latches and make full use of the incorporated logic blocks. This is in stark contrast to other synthesis methods we have studied where, on average, 46 percent of the total logic blocks used are of the latch only variety.
3. Our finite field multiplier designs are, on average, 48 percent smaller than designs produced by competing methods. In some cases, they are 60 to 65 percent smaller.

4. Our finite field multiplier designs are, on average, 20 percent faster than designs produced by competing methods. In some cases, they are 25 percent faster. This increase is due to a number of factors such as smaller overall designs and more regular layouts that contribute directly to shorter critical data paths.

In summary, pipelined finite field multipliers synthesized by our method demonstrate significant advantages over those synthesized by competing methods. Our circuits are better balanced and demonstrate reduced area and performance variability across finite fields of equivalent size. They are also, on average, 20 percent faster and 48 percent smaller. In some cases, our method produces pipelined multipliers that are up to 20 percent faster and up to 60 percent smaller than those synthesized by comparable means.

## 6 Conclusions

The efficient implementation of finite field multipliers on FPGAs is an enabling technology with respect to Software Defined Radios, Dynamic Cryptography, and other applications that implement finite field algorithms and require the speed and flexibility inherent in FPGAs. We have developed a novel logic synthesis algorithm and an overall automated design method for generating FPGA-based finite field multipliers that are smaller and faster than those obtained through competing and comparable methods. Our design method exploits the unique characteristics of the FPGA finite field multiplier design challenge. These characteristics include FPGA based logic synthesis, multiple-output equations expressed as Exclusive-or Sum of Products, and the insertion of performance improvement measures such as pipelining. We have implemented our design ideas in a complete suite of design tools that can design, simulate, and test pipelined and combinatorial finite field multipliers based on the generator polynomial for the field. Our resulting designs outperform those generated by competing methods in terms of both area and performance. In many cases, our designs are fifty to sixty percent smaller and between ten and twenty percent faster than designs synthesized with competing methods.

## 7 References

- [1] G. Ahlquist, B. Nelson, and M. Rice. Optimal finite field multipliers for FPGAs. In J. Irvine P. Lysaght and R. Hartenstein, editors, *9th International Workshop on Field Programmable Logic and Applications (FPL 99)*, pages 51–61, Glasgow, UK, Aug 1999. Springer-Verlag.
- [2] J. Bard. Software defined radios. Short Course Sponsored by Technology Training Corporation, May 2002. Notes and Commentary Provided by Dr John Bard as Part of His Software Defined Radios Course.
- [3] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.
- [4] S. MacLaird. Joint tactical radio system (JTRS). In *GEIA 2001 Standardization Symposium*. Government Electronics and Information Technology Association GEIA, November 2001.
- [5] R. J. McEliece. *Finite Fields for Computer Scientists and Engineers*. Kluwer Academic Publishers, 1987.
- [6] Rajeev Murgai, Robert K Brayton, and Alberto Sangiovanni-Vincentelli. *Logic Synthesis for Field Programmable Gate Arrays*. Kluwer Academic Publishers, 1995.