PAPER Special Section on Multiple-Valued Logic and VLSI Computing

Automatic Generation System for Multiple-Valued Galois-Field Parallel Multipliers*

Rei UENO[†a)], ***Student Member***, **Naofumi HOMMA**[†], ***and*** **Takafumi AOKI**[†], ***Members***

SUMMARY This paper presents a system for the automatic generation of Galois-field (GF) arithmetic circuits, named the *GF Arithmetic Module Generator* (*GF-AMG*). The proposed system employs a graph-based circuit description called the *GF Arithmetic Circuit Graph* (*GF-ACG*). First, we present an extension of the GF-ACG to handle $GF(p^m)$ ($p \geq 3$) arithmetic circuits, which can be efficiently implemented by multiple-valued logic circuits in addition to the conventional binary circuits. We then show the validity of the generation system through the experimental design of $GF(p^m)$ multipliers for different p-values. In addition, we evaluate the performance of three types of $GF(2^m)$ multipliers and typical $GF(p^m)$ multipliers ($p \geq 3$) empirically generated by our system. We confirm from the results that the proposed system can generate a variety of GF parallel multipliers, including practical multipliers over $GF(p^m)$ having extension degrees greater than 128.

key words: *GF arithmetic circuits, formal design, parallel multipliers, automatic generation, multiple-valued logic*

1. Introduction

Applications of error correction code (ECC) and cryptography based on arithmetic operations over Galois fields (GFs) are rapidly proliferating as the importance of reliable and secure communications increases [2] Recently, these operations are being increasingly implemented on hardware in embedded devices such as cell phones and radio-frequency identification (RFID) chips, and the performance of the arithmetic circuits have a significant impact on the effectiveness and security of the entire system. In addition, some applications with elliptic curve and pairing-based cryptographies employ GF arithmetic circuits of characteristic greater than 2 (i.e., $GF(p^m)$ where p and m are prime and natural numbers, respectively), which are inherently represented in a non-binary manner. For example, some pairing-based cryptographies use GF with $p = 3$ [2]–[6], and a hyperelliptic curve over GF with $p = 5$ or 7 is useful for efficient implementation of pairing-based cryptography [7], [8].

Most such arithmetic circuits have been designed at the lowest logic level by designers whose training in GF arithmetic is specialized for a particular type of application. Conventional hardware description languages (HDLs) do not currently have high-level arithmetic data structures, arithmetic operations, or formulae over Galois fields. Moreover, conventional high-level synthesis techniques have a difficulty in describing GF arithmetic circuits because the mapping of arithmetic operations over GFs varies with the basis and the modular polynomial even if they are represented by the same operator. Due to the large variety of GF multipliers, designers would be forced to describe the behavior of a GF multiplier in lowest-level expressions even with high-level synthesis tools. Furthermore, complete functional verification of GF arithmetic circuits designed by hand is much harder than that for integer arithmetic circuits because many GF operations in ECCs and cryptography are performed with operands having more than 64 bits for achieving enough error correction capability and resistance to cryptoanalysis attacks, respectively. Even for statistical verification by Monte Carlo simulation, the generation of test patterns for GF operations is more intractable than for integer operations.

To address the above problems, we present a system for the automatic generation of GF arithmetic circuits, named the *GF Arithmetic Module Generator* (*GF-AMG*). Given a circuit specification, the system generates the corresponding HDL description whose function is completely verified in a formal manner. The system we have developed focuses on the generation of $GF(2^m)$ and $GF(3^m)$ parallel multipliers for various modular polynomials. The basic idea of GF-AMG is to use a graph-based representation of GF arithmetic circuits, called the *GF Arithmetic Circuit Graph* (*GF-ACG*) [9], as the internal data structure. We can represent an arbitrary GF arithmetic circuit by a GF-ACG and verify it formally even if the operand bit length (i.e., the extension degree) is greater than 128. The verified GF-ACG is then mapped into the corresponding HDL description, which can be implemented in either multiple-valued logic or conventional binary logic.

In this paper, we first present an extension of the GF-ACG for describing arithmetic circuits over $GF(p^m)$ in order to design hardware for elliptic curve and pairing-based cryptographies including the above ones, and mapping them into multiple-valued logic circuits in addition to binary logic circuits. The basic idea of the extension is to add an encoding function from a p-valued variable to several R-valued variables ($p > R$) as new nodes at the lowest level, where R is determined by an implementation logic. The encoding function is given by algebraic equations. We then present our GF-AMG framework based on the extended GF-ACG

Manuscript received October 13, 2016.
Manuscript revised February 20, 2017.
Manuscript publicized May 19, 2017.

[†]The authors are with Tohoku University, Sendai-shi, 980–8579 Japan.

*A preliminary version of this paper appeared in the IEEE 45th International Symposium on Multiple-Valued Logic (ISMVL 2015) [1].

a) E-mail: ueno@aoki.ecei.tohoku.ac.jp

DOI: 10.1587/transinf.2016LOP0010

and describe our experimental evaluation of the system.

While a preliminary evaluation was performed in the previous version [1], in this paper, we show an extension and further evaluation of the proposed system. First, we extend our system to support more variety of $GF(p^m)$ multipliers while the previous version supported only $p = 2$ and 3. We then demonstrate the validity of GF-AMG using the experimental design and verification of $GF(p^m)$ ($p = 2, 3, 5, 7,$ and 11) parallel multipliers. The results show that the proposed system can generate a verified $GF(11^{256})$ multiplier in about five minutes. In addition, we show the generation results of typical $GF(2^m)$ parallel multipliers based on three arithmetic algorithms. We also show the generation results of typical multipliers over $GF(p^m)$ ($p \geq 3$) implemented in binary logic.

2. Galois-Field Arithmetic Circuit Graph

2.1 Definition of Galois-Field Arithmetic Circuit Graph

Figure 1 shows an overview of the GF-ACG. A GF-ACG G is defined as $(\boldsymbol{N}, \boldsymbol{E})$, where $\boldsymbol{N}$ is a set of nodes and $\boldsymbol{E}$ is a set of directed edges. A node represents an arithmetic circuit by its functional assertion and internal structure. A directed edge represents the flow of data between nodes and defines the data dependency. We assume that every node has at least one edge connection.

A node $n \in \boldsymbol{N}$ is defined by $(\boldsymbol{F}, G')$, where $\boldsymbol{F}$ is the functional assertion given as a set of equations over GFs (the *GF equations*) and G' is the internal structure given as a smaller GF-ACG. A node at the lowest level of abstraction does not have an internal structure and is thus described as $(\boldsymbol{F}, nil)$. A functional assertion is represented as a relation $E_l = E_r$, where E_l and E_r are the output and input expressions, respectively, and each expression is given by variables, constants, or combinations of two or more expressions connected by any of the arithmetic operators $+$, $-$, or $\times$.

A directed edge $e \in \boldsymbol{E}$ is defined as $(src, dest, x)$, where *src* and *dest* represent the start and end nodes, respectively, and x represents the variable indicating an element of GF. If either *src* or *dest* is *nil*, the directed edge represents an external input or output for the given GF-ACG. Each variable x is associated with a Galois field. A Galois field GF is defined as $(\boldsymbol{B}, \boldsymbol{C}, IP)$, where $\boldsymbol{B}$ is the basis, $\boldsymbol{C}$ is the coefficient vector, and IP is the irreducible polynomial. More precisely, $\boldsymbol{B}$, $\boldsymbol{C}$, and IP are given as

$$\boldsymbol{B} = (\gamma_{m-1}, \gamma_{m-2}, \ldots, \gamma_i, \ldots, \gamma_0), \tag{1}$$

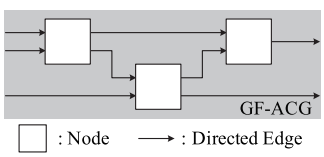

Fig. 1 Overview of Galois-field arithmetic circuit graph.

$$\boldsymbol{C} = (C_{m-1}, C_{m-2}, \ldots, C_i, \ldots, C_0), \tag{2}$$

$$IP = \beta^m + c_{m-1}\beta^{m-1} + \cdots + c_i\beta^i \cdots + c_0\beta^0, \tag{3}$$

where β is the indeterminate element (i.e., a root of an irreducible polynomial), m is the degree of field extension, C_i is the coefficient set of degree i ($i \in \mathbb{Z}$, $0 \leq i \leq m-1$), and c_i is the i-th element of the coefficient set C_i. $\gamma_i = \beta^i$ if GF is represented by a polynomial basis (PB), and $\gamma_i = \alpha^{p^i}$ if GF is represented by a normal basis (NB), where $\alpha = \beta^n$ and p is the characteristic. $IP = nil$ if GF is a prime field. Thus, the above description can handle both prime and extension fields. Let h ($0 \leq h \leq m-1$) and l ($0 \leq l \leq h$) be the most and least significant degrees, respectively. A variable is then represented as $x = (GF, (h, l))$, where the ordered pair (h, l) is called the degree range. Using the above notation, we can handle any specific variable x_i of degree i.

A variable can be decomposed to an expression with sub-variables at a lower level of abstraction. Let x be a variable and x_i ($l \leq i \leq h$) be a lower-level variable. We have two types of decomposition nodes, whose functions are given as

$$x_h^{(e)} + x_{h-1}^{(e)} + \cdots + x_l^{(e)} = x, \tag{4}$$

$$x_h^{(p)}\gamma_h + x_{h-1}^{(p)}\gamma_{h-1} + \cdots + x_l^{(p)}\gamma_l = x. \tag{5}$$

Equation (4) indicates that $x \in GF(p^m)$ is divided into a number of variables of degree i [i.e., $x_i^{(e)} \in GF(p^m)$, $l \leq i \leq h$]. On the other hand, Eq. (5) indicates that $x \in GF(p^m)$ is divided into a number of variables over the prime field [i.e., $x_i^{(p)} \in GF(p)$, $l \leq i \leq h$]. We also have two types of composition nodes, given as the inverse relations of the above inputs and outputs. Using the decomposition and composition nodes, we can change the level of abstraction of edge representation. Note that these nodes are implemented by wiring and have no internal structures.

The above GF-ACG can also be used to represent any binary logic circuit. A logic variable is defined as a variable over the GF whose coefficient set is limited to the zero element "0" and the unit element "1". Any binary logic operation can be represented with pseudo-logic equations, such as $and(a, b) = ab$. Note that the idempotent law is defined as one of the functional assertions in the corresponding node (i.e., $a = a^2$ and $b = b^2$).

Thus, the original GF-ACG can represent any $GF(2^m)$ arithmetic circuit in binary logic. The arithmetic circuits given by GF-ACGs are verified by a formal verification method using a Gröbner basis and a polynomial reduction technique [9].

2.2 Extension to $GF(p^m)$ Arithmetic Circuit

An extension of the GF-ACG is presented for describing a $GF(p^m)$ arithmetic circuit, enabling it to be implemented in multiple-valued logic as well as in binary logic. In the above GF-ACG, a mapping from a GF variable to a logic variable at the lowest level description is implicitly given [i.e., 0 and 1 in $GF(2)$ are mapped into 0 and 1 in binary logic, respectively] because it focuses only on $GF(2^m)$ arithmetic circuits

Table 1 Mapping of GF values onto logic values.

(a) An example of $GF(2)$

$GF(2)$ value	Logic value
0	0
1	1

(b) An example of $GF(3)$

$GF(3)$ value	Logic value
0	00
1	01
2	10

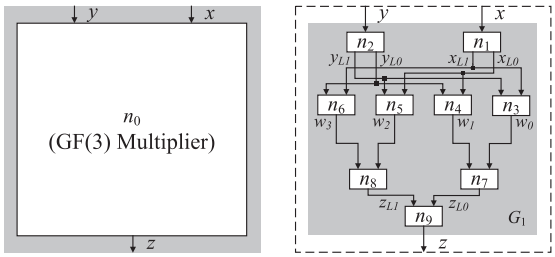

Fig. 2 GF-ACGs for $GF(3)$ multiplier.

Table 2 Nodes, Galois fields and variables in Fig. 2 (b).

Nodes
$n_0 = (\{z = x \times y\}, G_1)$
$n_1 = (\{(1 - x_{L1})x_{L0} + 2x_{L1}(1 - x_{L0}) = x, x_{L0}x_{L1} = 0\}, nil)$
$n_2 = (\{(1 - y_{L1})y_{L0} + 2y_{L1}(1 - y_{L0}) = y, y_{L0}y_{L1} = 0\}, nil)$
$n_3 = (\{w_0 = AND(x_{L1}, y_{L1})\}, nil)$
$n_4 = (\{w_1 = AND(x_{L0}, y_{L0})\}, nil)$
$n_5 = (\{w_2 = AND(x_{L0}, y_{L1})\}, nil)$
$n_6 = (\{w_3 = AND(x_{L1}, y_{L0})\}, nil)$
$n_7 = (\{z_{L0} = OR(w_0, w_1)\}, nil)$
$n_8 = (\{z_{L1} = OR(w_2, w_3)\}, nil)$
$n_9 = (\{z = (1 - z_{L1})z_{L0} + 2z_{L1}(1 - z_{L0}), z_{L1}z_{L0} = 0\}, nil)$

Galois field
$GF(3) = ((\beta^0), (\{0, 1, 2\}), nil)$
$Logic = ((\beta^0), (\{0, 1\}), nil)$

Galois field variables	
$x = (GF(3), (0, 0))$	$x_{L0}, x_{L1} = (Logic, (0, 0))$
$y = (GF(3), (0, 0))$	$y_{L0}, y_{L1} = (Logic, (0, 0))$
$z = (GF(3), (0, 0))$	$z_{L0}, z_{L1} = (Logic, (0, 0))$
$w_i = (Logic, (0, 0)), (0 \le i \le 3)$	

and their binary implementations. Therefore, such mapping is done without the need for any additional procedure. In order to describe and verify nodes with $GF(p)$ ($p \ge 2$) variables and their R-valued ($R \ge 2$) implementation, however, we need to give an explicit mapping at the lowest level of abstraction. Our plan is to provide a mapping function, called an encoding function, for transforming $GF(p)$ variables into R-valued logic variables in the form of a functional assertion (i.e., a GF equation) for the lowest-level nodes.

We first describe an encoding function for transforming $GF(p)$ variables into binary logic variables. Each GF variable in C_i (a coefficient set of degree i) is encoded by at least $\lceil \log_2 |C_i| \rceil$ logic variables. Table 1 gives examples of such mappings, showing encodings of (a) $GF(2) \in \{0, 1\}$ and (b) $GF(3) \in \{0, 1, 2\}$ into binary logic variables. Note that for cases having characteristic $p > 2$, any encoding is possible, including non-minimum-length encoding. Such encoding can be represented by a specific equation, referred to as an *encoding equation*. Let x and L_j ($0 \le j \le k - 1$) be a GF variable over $GF(p)$ and a logic variable used for encoding, respectively. Let $\boldsymbol{\alpha} = (\alpha_0, \alpha_1, \ldots, \alpha_{k-1}) \in \{0, 1\}^k$ be a k-bit logic value. The general form of the encoding equation is then given as

$$x = \sum_{\boldsymbol{\alpha} \in \{0,1\}^k} \left(f(\boldsymbol{\alpha}) \times \Pi_{j=0}^{k-1} L_j^{\alpha_j} \right), \tag{6}$$

where $f(\boldsymbol{\alpha})$ is the GF value corresponding to $\boldsymbol{\alpha}$, and $L_j^{\alpha_j}$ is the j-th literal, defined as

$$L_j^{\alpha_j} = \begin{cases} 1 - L_j & (\alpha_j = 0) \\ L_j & (\alpha_j = 1) \end{cases}. \tag{7}$$

For example, the encoding equations for Table 1 (a) and (b) are given as $x = L_0$ and $x = (1 - L_1)L_0 + 2L_1(1 - L_0)$, respectively.

Figure 2 shows GF-ACGs for a 2-input multiplier over $GF(3)$ implemented in binary logic [10], where the node in Fig. 2 (a) corresponds to the shaded part in Fig. 2 (b). This indicates that node n_0 has an internal structure consisting of lower-level nodes in the corresponding shaded part. Table 2 shows the nodes, GFs and variables used in Fig. 2. The nodes of n_1, n_2, and n_9 in Fig. 2 (b) perform the mapping between GF variables and logic variables. More precisely, the functions of n_1 and n_2 are to translate GF variables into logic variables, while the function of n_9 is to translate logic variables into GF variables. Note that the functional assertions of such nodes require equation(s) that represent unused inputs. In this example, one such equation is given as $x_{L0}x_{L1} = 0$ because $(x_{L0}, x_{L1}) = (1, 1)$ is not used. Thus, any $GF(p^m)$ arithmetic circuit will be implemented by binary logic circuits in a uniform manner.

Next, we then describe an extension of the above encoding equation for the case of R-valued implementation. Table 3 shows examples of the mapping of (a) GF(3) $\in \{0, 1, 2\}$ and (b) GF(5) $\in \{0, 1, 2, 3, 4\}$ into ternary logic. Let x and L_j ($0 \le j \le k-1$) be a GF variable over $GF(p)$ and an R-valued logic variable used for encoding, respectively. Let $\boldsymbol{\alpha} = (\alpha_0, \alpha_1, \ldots, \alpha_{k-1}) \in \{0, 1, \ldots, R-1\}^k$ be a k-bit R-valued logic value; the encoding equation is then given as

$$x = \sum_{\boldsymbol{\alpha} \in \{0,1,\ldots,R-1\}^k} \left(f(\boldsymbol{\alpha}) \times \Pi_{j=0}^{k-1} L_j^{\alpha_j} \right), \tag{8}$$

and $L_j^{\alpha_j}$ is represented by

$$L_j^{\alpha_j} = \prod_{\substack{l \in \{0,1,\ldots,R-1\} \\ l \ne \alpha_j}} \frac{L_j - l}{\alpha_j - l}, \tag{9}$$

where Eqs. (8) and (9) are based on arithmetic operations over $GF(p)$.

For example, the encoding equations for Table 3 (a) and (b) are given by $x = L_0$ and $x = (2L_1 + 3L_1 + 1)L_0 + 2L_1^2 + L_1$, respectively.

Table 3 Mapping of GF values onto 3-valued logic values.

(a) An example of $GF(3)$

$GF(3)$ value	3-valued logic value
0	0
1	1
2	2

(b) An example of $GF(5)$

$GF(5)$ value	3-valued logic value
0	00
1	01
2	02
3	10
4	11

3. Galois-Field Arithmetic Module Generator

This section presents an automatic generation system, the Galois-Field Arithmetic Module Generator (GF-AMG), for producing GF parallel multipliers. The system employs the extended GF-ACG for producing multiplier modules whose functions are completely verified at the algorithmic level.

3.1 System Framework

Figure 3 is a block diagram of GF-AMG. It consists of (i) the GF-ACG Code Synthesizer, (ii) the GF-ACG Verifier, and (iii) the ACG-to-HDL Translator. The GF-ACG Code Synthesizer generates GF-ACG code according to the user's design specification, which includes characteristic, multiplication algorithm, modular polynomial, and logic type. Table 4 shows a list of characteristics, multiplication algorithms, modular polynomial degrees, and logic types that can be generated by the GF-AMG system. The GF-ACG Verifier proceeds to formally verify the generated GF-ACG code by a method using a Gröbner basis and a polynomial reduction technique, following the procedure given in [9]. The ACG-to-HDL Translator then translates the verified GF-ACG code into the equivalent Verilog-HDL code, using the algorithm shown in Algorithm 1. Given a GF-ACG G, we extract a set of relations of internal edges at the lowest level of abstraction from G recursively. The relations of internal edges are then translated into the corresponding HDL format by one-to-one mapping.

3.2 Generation of $GF(p^m)$ Parallel Multipliers

This section focuses on the design and generation of $GF(p^m)$ parallel multipliers in GF-AMG. For the conventional design of $GF(2^m)$ parallel multipliers also generated by GF-AMG, see [11], [12], and [13]. Let x and $y \in GF(p^m)$ be the inputs and let $z \in GF(p^m)$ be the output. The multiplication over $GF(p^m)$ is first divided into the following two functions:

$$\sum_{i=0}^{m-1} w_i = x \times y, \tag{10}$$

$$z = \sum_{i=0}^{m-1} w_i, \tag{11}$$

where $w_i \in GF(p^m)$ $(0 \le i \le n-1)$ is the i-th partial product. We then consider the internal structure of the nodes

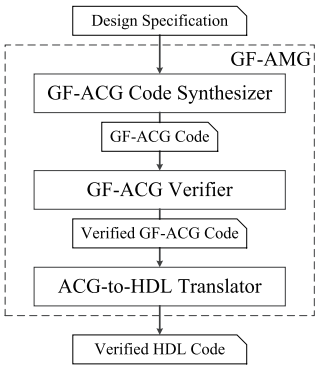

Fig. 3 Block diagram of GF-AMG.

Table 4 Specification supported by GF-AMG.

Characteristic p	Algorithm	Degree for IP	Logic type
2	Full-tree	2-256	binary
	Mastrovito	2-256	
	Massey-Omura	2-64	
3, 5, 7, 11	Full-tree	2-256	binary p-valued logic

Algorithm 1 Translate GF-ACG to HDL

Input: GF-ACG $G = (\boldsymbol{N}, \boldsymbol{E})$
Output: HDL Description $\boldsymbol{D}$

```
function MAPPING(G)
    D' := ∅; str S;
    for each (F, G') ∈ N do
        if G' ≠ nil then
            D' := D' ∪ Mapping(G');
        end if
    end for
    S := GFACGtoHDLmodule(G);
    D := D' ∪ {S};
    return D
end function
```

corresponding to Eqs. (10) and (11) to obtain its hierarchical GF-ACG description. Each w_i is given by

$$w_i = x_i \times y,\ 0 \le i \le m-1 \tag{12}$$

where x_i is the i-th element obtained by the decomposition of $x = \sum_{i=0}^{m-1} x_i$ and $y \in GF(p^m)$. This means that the internal structure of the node performing Eq. (10) is composed of m nodes performing Eq. (12). The nodes corresponding to Eq. (11) are composed of $m-1$ 2-input 1-output adders over $GF(p^m)$, which are given by m 2-input 1-output adders over $GF(p)$.

For example, Fig. 4 shows the GF-ACGs for the $GF(3^4)$ parallel multiplier at the top four levels of abstraction. Table 5 shows the corresponding nodes, GFs and GF variables. Note that the decomposition and composition nodes are not shown in Table 5. The nodes in Fig. 4 (a), (b), and (c) correspond to the shaded parts in Fig. 4 (b), (c), and (d), respectively. The 2^{nd}-level nodes "Partial Product Generator" (PPG) and GF "Accumulator" (GFA) in Fig. 4 (b) have functional assertions corresponding to Eqs. (10) and (11), respectively. The 3^{rd}-level nodes "PPGi" in Fig. 4 (c)

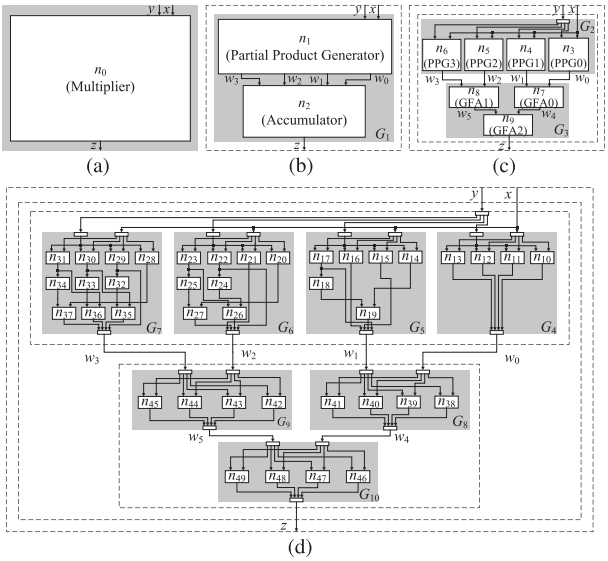

Fig. 4 GF-ACGs for $GF(3^4)$ parallel multipliers of (a) the top-level to (d) the 4^{th}-level.

have the functional assertion corresponding to Eq. (12). The nodes "GFAi" in Fig. 4 (c) indicate 2-input 1-output adders over $GF(3^4)$ to construct "Accumulator". In addition, the nodes in Fig. 4 (d) indicate $GF(3)$ arithmetic circuits, and these are described as given in Fig. 2, which showed the binary implementation case. Thus, we have the GF-ACGs for the $GF(p^m)$ parallel multiplier represented in a hierarchical manner.

Algorithm 2 displays an algorithm for synthesizing $GF(p^m)$ multipliers. Given a design specification (i.e., an irreducible polynomial and an implementation logic), the algorithm generates a GF-ACG. The function "Degree" in Line 2 obtains the degree of the irreducible polynomial. According to the value obtained, its internal structure is generated in a recursive manner. The function "GetEquation" in Line 4 obtains an equation of the "PPGi" expressions that are represented in Eq. (12). The functions "CountSubOperator" and "CountAddOperator" count the numbers of "$-$" and "$+$" operators in the equation, respectively. Using the above numbers and the degree, we generate 4^{th}-level GF-ACGs for $GF(p)$ arithmetic circuits. The functions "GenerateGFpMultiplier", "GenerateGFpAdditiveInv", and "GenerateGFpAdder" return GF-ACGs for $GF(p)$ multipliers, additive inverters, and adders, respectively. Their internal structures are determined by the given logic L. If L is binary logic, the internal structure is given by the netlist corresponding to $GF(p)$ multiplier, additive inverter, and adder which are designed in a manner similar to Fig. 2. If L is not binary logic, the internal structure is given as *nil* in order for designers to use custom logic cells designed by themselves. The function "GeneratePPGi" in Line 16 generates "PPGi" expressions ($0 \le i \le d-1$) from the 4^{th}-level GF-ACGs, where the $GF(p)$ adders are placed as a tree. The function "GeneratePPG" in Line 18 generates a GF-ACG for "Partial Product Generator" from the 3^{rd}-level GF-ACGs of "PPGi". Similarly, "Accumulator" is generated from the 3^{rd}-level GF-ACGs of "GFAi" consisting of d GF-ACGs of $GF(p)$ adders. In "Accumulator", $d-1$ "GFAi" expressions are placed as a tree. Finally, the function "GenerateMultiplier" in Line 26 generates a GF-ACG for the $GF(p^m)$ multiplier from the 2^{nd}-level GF-ACG.

The HDL code generated for $GF(p^m)$ multipliers may be applied in not only a binary implementation but also an

Table 5 Nodes, Galois fields and variables in Fig. 4.

Nodes
[Multiplier] $n_0 = (\{z = x \times y\}, G_1)$
[Partial Product Generator]
$n_1 = (\{w_0 + w_1 + w_2 + w_3 = x \times y\}, G_2)$
[PPG0] $n_3 = (\{w_0 = x \times y_0\}, G_4)$
$n_{10} = (\{w_{0,0} = x_0 \times y_{0,0}\}, G_{11})$
$n_{11} = (\{w_{0,1} = x_1 \times y_{0,0}\}, G_{12})$
$n_{12} = (\{w_{0,2} = x_2 \times y_{0,0}\}, G_{13})$
$n_{13} = (\{w_{0,3} = x_3 \times y_{0,0}\}, G_{14})$
[PPG1] $n_4 = (\{w_1 = x \times y_1\}, G_5)$
$n_{14} = (\{w_6 = x_0 \times y_{1,1}\}, G_{15})$
$n_{15} = (\{w_{1,2} = x_1 \times y_{1,1}\}, G_{16})$
$n_{16} = (\{w_{1,3} = x_2 \times y_{1,1}\}, G_{17})$
$n_{17} = (\{w_{1,0} = x_3 \times y_{1,1}\}, G_{18})$
$n_{18} = (\{w_7 = -w_{1,0}\}, G_{19})$
$n_{19} = (\{w_{1,1} = w_6 + w_7\}, G_{20})$
[PPG2] $n_5 = (\{w_2 = x \times y_2\}, G_6)$
$n_{20} = (\{w_8 = x_0 \times y_{2,2}\}, G_{21})$
$n_{21} = (\{w_{2,3} = x_1 \times y_{2,2}\}, G_{22})$
$n_{22} = (\{w_{2,0} = x_2 \times y_{2,2}\}, G_{23})$
$n_{23} = (\{w_9 = x_3 \times y_{2,2}\}, G_{24})$
$n_{24} = (\{w_{10} = -w_{2,0}\}, G_{25})$
$n_{25} = (\{w_{11} = -w_9\}, G_{26})$
$n_{26} = (\{w_{2,1} = w_{10} + w_9\}, G_{27})$
$n_{27} = (\{w_{2,2} = w_8 + w_{11}\}, G_{28})$
[PPG3] $n_6 = (\{w_3 = x \times y_3\}, G_7)$
$n_{28} = (\{w_{12} = x_0 \times y_{3,3}\}, G_{29})$
$n_{29} = (\{w_{3,0} = x_1 \times y_{3,3}\}, G_{30})$
$n_{30} = (\{w_{13} = x_2 \times y_{3,3}\}, G_{31})$
$n_{31} = (\{w_{14} = x_3 \times y_{3,3}\}, G_{32})$
$n_{32} = (\{w_{15} = -w_{3,0}\}, G_{33})$
$n_{33} = (\{w_{16} = -w_{13}\}, G_{34})$
$n_{34} = (\{w_{17} = -w_{14}\}, G_{35})$
$n_{35} = (\{w_{3,1} = w_{13} + w_{15}\}, G_{36})$
$n_{36} = (\{w_{3,2} = w_{14} + w_{16}\}, G_{37})$
$n_{37} = (\{w_{3,3} = w_{12} + w_{17}\}, G_{38})$
[Accumulator]
$n_2 = (\{z = w_0 + w_1 + w_2 + w_3\}, G_3)$
[GFA0] $n_7 = (\{w_4 = w_0 + w_1\}, G_8)$
$n_{38+i} = (\{w_{4,i} = w_{0,i} + w_{1,i}\}, G_{39+i}),\ (0 \le i \le 3)$
[GFA1] $n_8 = (\{w_5 = w_2 + w_3\}, G_9)$
$n_{42+i} = (\{w_{5,i} = w_{2,i} + w_{3,i}\}, G_{43+i}),\ (0 \le i \le 3)$
[GFA2] $n_9 = (\{z = w_4 + w_5\}, G_{10})$
$n_{46+i} = (\{z_i = w_{4,i} + w_{5,i}\}, G_{47+i}),\ (0 \le i \le 3)$

Galois field
$GF(3^4) = ((\beta^3, \beta^2, \beta^1, \beta^0),$
$(\{0,1,2\}, \{0,1,2\}, \{0,1,2\}, \{0,1,2\}), \beta^4 + \beta + 2)$
$GF(3) = ((\beta^0), (\{0,1,2\}), nil)$

Galois field variables
$x, y, z = (GF(3^4), (3,0))$
$x_i, y_i, z_i = (GF(3), (0,0)),\ (0 \le i \le 3)$
$y_{i,i} = (GF(3), (0,0)),\ (0 \le i \le 3)$
$w_j = (GF(3^4), (3,0)),\ (0 \le j \le 5)$
$w_{j,k} = (GF(3), (0,0)),\ (0 \le j \le 5,\ 0 \le k \le 3)$

Table 6 Generation times of multipliers over $GF(p^m)$ (sec).

Extended degree m	8					16					32				
Characteristic p	2	3	5	7	11	2	3	5	7	11	2	3	5	7	11
GF-ACG synthesis	0.07	0.07	0.07	0.07	0.07	0.08	0.08	0.08	0.09	0.09	0.12	0.14	0.14	0.15	0.15
Formal verification	2.59	2.78	2.77	2.78	2.77	4.06	4.26	4.24	4.25	4.24	7.40	7.70	7.62	7.70	7.66
ACG-to-HDL	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.02	0.02	0.02	0.10	0.11	0.11	0.11	0.11
Total	2.66	2.86	2.84	2.86	2.85	4.17	4.37	4.35	4.36	4.36	7.62	7.95	7.87	7.96	7.92

Extended degree m	64					128					256				
Characteristic p	2	3	5	7	11	2	3	5	7	11	2	3	5	7	11
GF-ACG synthesis	0.29	0.35	0.44	0.37	0.39	0.99	1.23	1.30	1.33	1.39	3.90	4.99	5.18	5.32	5.58
Formal verification	16.39	17.26	18.37	17.25	17.26	48.25	54.14	54.09	54.08	54.22	235.62	289.79	293.42	292.79	292.34
ACG-to-HDL	0.58	0.66	0.81	0.66	0.67	3.63	4.63	4.33	4.56	4.40	27.57	33.05	32.88	33.34	33.33
Total	17.26	18.27	19.62	18.28	18.32	52.87	60.00	59.72	59.97	60.01	267.09	327.83	331.48	331.45	331.24

Algorithm 2 Synthesize GF-ACG

Input: Irreducible Polynomial IP, Logic L
Output: GF-ACG $G = (N, E)$

function FULLTREE($IP, Logic$)
  d := Degree(IP); **str** eq;
  for $i = 0$ to $d - 1$ **do**
    eq := GetEquation(i, IP);
    l := CountSubOperator(eq);
    m := CountAddOperator(eq);
    for $j = 0$ to $d - 1$ **do**
      G_{d+j} := GenerateGFpMultiplier(L); ▷ 4^{th}-level
    end for
    for $j = 0$ to $l - 1$ **do**
      G_{2d+j} := GenerateGFpAdditiveInv(L); ▷ 4^{th}-level
    end for
    for $j = 0$ to $m - 1$ **do**
      G_{2d+l+j} := GenerateGFpAdder(L); ▷ 4^{th}-level
    end for
    G_i := GeneratePPGi($eq, G_d, \ldots, G_{2d+l+m-1}$); ▷ 3^{rd}-level
  end for
  G' := GeneratePPG($G_0, \ldots, G_{d-1}$); ▷ 2^{nd}-level
  for $i = 0$ to $d - 2$ **do**
    for $j = 0$ to $d - 1$ **do**
      G_{d+j-1} := GenerateGFpAdder(L); ▷ 4^{th}-level
    end for
    G_i = GenerateGFAi($G_{d-1}, \ldots, G_{2d-2}$); ▷ 3^{rd}-level
  end for
  G'' = GenerateACC($G_0, \ldots, G_{d-2}$); ▷ 2^{nd}-level
  G = GenerateMultiplier(G', G''); ▷ top-level
  return G
end function

L-valued implementation. [$GF(2^m)$ multipliers, by contrast, are implemented only in binary logic.] For a binary implementation, we can apply the HDL code to the standard back-end design flow including logic synthesis and placement and routing (P&R) with the standard cell library. For an L-valued implementation, we would implement the HDL code by a technology mapping with a custom-made library in an L-valued logic. Thus, we see that GF-AMG generates verified HDL codes for both multiple-valued logic and binary logic.

As an example, Fig. 5 shows a schematic of a $GF(3^8)$ multiplier generated by GF-AMG, where the lowest level component indicates an arithmetic circuit over $GF(3)$. We can implement this multiplier in ternary logic by applying a ternary logic circuit to the component.

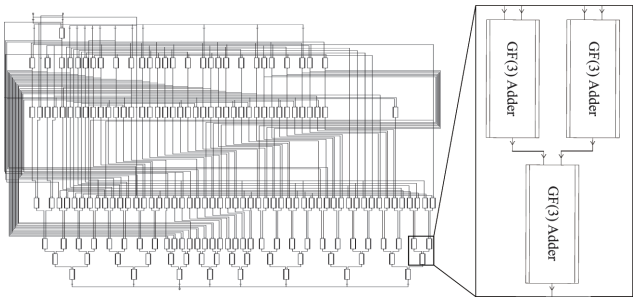

Fig. 5 Schematic of $GF(3^8)$ multiplier obtained from GF-AMG.

3.3 Experimental Generation

The performance of our system was evaluated through the experimental generation of $GF(p^m)$ parallel multipliers. We first generated a set of $GF(p^m)$ parallel multipliers of typical degrees. The generation was conducted with an open-source computer algebra software Risa/Asir [14] under Linux on a PC (an Intel Xeon E5450 with a 3.00-GHz processor and 32 GB of RAM).

Table 6 shows the generation times, consisting of GF-ACG synthesis, verification, and GF-ACG-to-HDL translation times, for each of the degrees investigated. Using our method, we achieved complete verification even for a 1024-bit multiplier over $GF(11^{256})$. Note here that the verification time decreases even in the case of a larger p because the computation time of algebraic operation with the software used in the experiment is sometimes dependent on the machine condition such as parallel-executed processes. As a comparison to evaluate the advantage of the verifier, we also performed the Verilog-XL simulation using the corresponding HDL descriptions. With this method, we were not able to complete the simulation of $GF(3^{10})$ or larger multipliers because the simulation time increases exponentially as the extension degree increases. As described above, GFs with at most characteristic seven are used for pairing-based cryptography so far. Thus, the experimental result suggests that our system is sufficient and available for such applications.

We then generated a set of $GF(2^m)$ parallel multipliers for three types of multiplication algorithm in order to assess

Table 7 Performance of $GF(p^m)$ multipliers for different characteristics and degrees.

	Area (KGates)						Delay (ns)					
Degree m	4	8	16	32	64	128	4	8	16	32	64	128
$p = 3$	0.6	2.3	9.7	39.3	158.1	685.8	1.48	2.36	2.81	3.25	3.68	4.13
$p = 5$	2.22	9.67	40.28	164.34	663.78	2,667.98	2.83	3.74	4.63	5.54	6.43	7.34
$p = 7$	5.26	23.02	96.71	399.90	1,572.58	N/A	5.28	6.58	9.09	9.24	11.78	N/A
$p = 11$	14.06	61.99	259.32	1,043.41	4,247.72	N/A	9.95	12.44	17.09	19.43	21.88	N/A

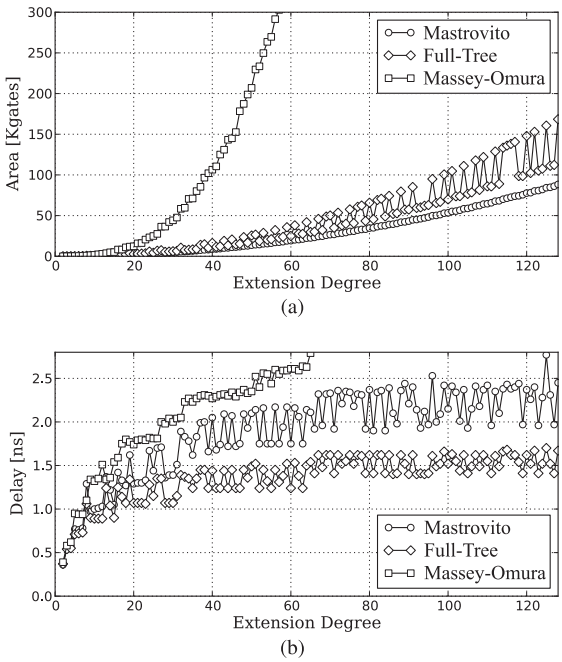

Fig. 6 Comparison of three types of $GF(2^m)$ multiplier for different extension degrees.

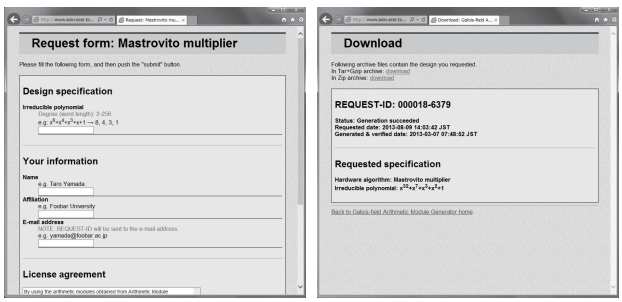

Fig. 7 GF-AMG Website: (a) request page and (b) download page.

the performance variation. The performance was evaluated with the Synopsys Design Compiler and the TSMC 65-nm cell library.

Figure 6 shows the area and delay of the three types of $GF(2^m)$ multiplier for different value of m, where the vertical axis indicates the (a) area or (b) delay, and the horizontal axis indicates the extension degree. We can confirm here that Mastrovito and Full-Tree have the advantage in area and delay, respectively. Massey-Omura, which is a typical multiplication algorithm using a normal basis (NB), did not demonstrate any advantage in area or delay over the other two algorithms. However, Massey-Omura is useful for more sophisticated arithmetic NB circuits; for example, we can design efficient exponential circuits based on an NB since the squaring operation is performed only by wiring.

Table 7 shows the performance of $GF(p^m)$ multipliers implemented in a binary logic, for different characteristics and degrees. For $GF(p^m)$ multipliers with $p = 5$, 7, and 11, we implemented $GF(p)$ arithmetic circuits (i.e., adder, multiplier, and constant multipliers over $GF(p)$) using the corresponding lookup-table. The Synopsys Design Compiler could not synthesize the $GF(7^{128})$ and $GF(11^{128})$ multipliers under our experimental condition due to the memory overflow. This would be because the circuit area (i.e., the number of logic gates) for the multipliers are too large ($\sim$ 6 M gates). However, the results suggest that designers can generate a variety of practical GF multipliers from given design specifications by the proposed GF-AMG system.

4. Conclusion

In this paper, we have presented a system named *GF-AMG* for the automatic generation of GF parallel multipliers that uses a graph-based circuit description called *GF-ACG*. We first extended the GF-ACG for $GF(p^m)$ ($p \geq 2$) arithmetic and R-valued ($R \geq 2$) implementation. We then showed the system framework of GF-AMG, wherein the generated HDL codes are completely verified by a formal verification method. We also evaluated the performance of GF-AMG by the experimental generation of $GF(p^m)$ multipliers. In particular, we demonstrated that the extended GF-ACG allows us to generate $GF(p^m)$ multipliers that can be implemented in multiple-valued logic.

The system described here will be available at our website [15], as depicted in Fig. 7. Designers can submit their specification on the request page [Fig. 7 (a)] and then receive the generated HDL code from the download page [Fig. 7 (b)].

Acknowledgments

We would like to show our greatest appreciation to Mr. Yukihiro Sugawara and Mr. Kotaro Okamoto for their valuable and insightful comments. This work has been supported by JSPS KAKENHI Grant No. 25240006.

References

[1] Y. Sugawara, R. Ueno, N. Homma, and T. Aoki, "System for automatic generation of parallel multipliers over Galois field," IEEE

45th International Symposium on Multiple-Valued Logic (ISMVL 2015), pp.54–59, 2015.
[2] E. Savas and C. Koc, "Finite field arithmetic for cryptography," IEEE Circuits Syst. Mag., vol.10, no.2, pp.40–56, Aug. 2010.
[3] I. Duursma and H.-S. Lee, "Tate pairing implementation for hyperelliptic curves $y^2 = x^p - x + d$," Advances in Cryptology—ASIACRYPT 2003, Lecture Notes in Computer Science, vol.2894, pp.111–123, Springer, 2003.
[4] P.S.L.M. Barreto, B. Lynn, and M. Scott, "Efficient implementation of pairing-based cryptosystems," Journal of Cryptology, vol.17, no.4, pp.321–334, 2004.
[5] D.J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," Journal of Cryptographic Engineering, vol.2, no.2, pp.77–89, 2012.
[6] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the Weil pairing," Journal of Cryptology, vol.17, no.4, pp.297–319, 2004.
[7] I. Duursma and K. Sakurai, "Efficient algorithms for the Jacobian variety of hyperelliptic curves $y^2 = x^p - x + 1$ over a finite field of odd characteristic p," Coding Theory, Cryptography and Related Areas, pp.73–89, Springer, 1998.
[8] E. Lee, H.-S. Lee, and Y. Lee, "Eta pairing computation on general divisors over hyperelliptic curves $y^2 = x^p - x + d$," Journal of Symbolic Computation, vol.43, no.6-7, pp.452–474, 2008.
[9] N. Homma, K. Saito, and T. Aoki, "A formal approach to designing cryptographic processors based on $GF(2^m)$ arithmetic circuits," IEEE Trans. Inf. Forensics and Security, vol.7, no.1, pp.3–13, Feb. 2012.
[10] D. Page and N.P. Smart, "Hardware implementation of finite fields of characteristic three," CHES 2002, Lecture Notes in Computer Science, vol.2523, pp.529–539, 2002.
[11] K. Okamoto, N. Homma, and T. Aoki, "A graph-based approach to designing parallel multipliers over Galois fields based on normal basis representations," Proc. 43th IEEE Int. Symp. Multiple-Valued Logic, pp.158–163, May 2013.
[12] A. Halbutogullari and C. Koc, "Mastrovito multiplier for general irreducible polynomials," IEEE Trans. Comput., vol.49, no.5, pp.503–518, May 2000.
[13] J. Massey and J. Omura, "Computational method and apparatus for finite field arithmetic," 1986. US Patent.
[14] "Risa/Asir (Kobe distribution) download page," http://www.math.kobe-u.ac.jp/Asir/asir.html
[15] "Arithmetic module generator for GF parallel multipliers," http://www.aoki.ecei.tohoku.ac.jp/arith/gfamg/

Rei Ueno received a B.E. degree in Information Engineering, and the M.S. degree in Information Sciences from Tohoku University, Sendai, Japan, in 2013 and 2015, respectively. He is currently enrolled in a doctorial course at Tohoku University. Since 2016, he has been a JSPS (The Japan Society for the Promotion of Science) research fellow. His research interests include arithmetic circuits, cryptographic implementations, formal verification, and hardware security.

Naofumi Homma received the B.E. degree in information engineering, and the M.S. and Ph.D. degrees in information sciences from Tohoku University, Sendai, Japan, in 1997, 1999 and 2001, respectively. He is currently a Professor of the Research Institute of Electrical Communication at Tohoku University. For 2002–2006, he also joined the Japan Science and Technology Agency (JST) as a researcher for the PRESTO project. His research interests include computer arithmetic, EDA methodology, high performance/secure VLSI computing, and hardware security. Dr. Homma received the IP Award at the 2005 LSI IP Design Award, the Best Paper Award at the Workshop on Synthesis And System Integration of Mixed Information Technologies in 2007, the Best Symposium Paper Award at the 2013 IEEE International Symposium on Electromagnetic Compatibility, and the Best Paper Award at the Workshop on Cryptographic Hardware and Embedded Systems 2014 (CHES 2014).

Takafumi Aoki received the B.E., and M.E., and D.E. degrees in electronic engineering from Tohoku University, Sendai, Japan, in 1988, 1990, and 1992, respectively. He is currently a Professor of the Graduate School of Information Sciences at Tohoku University. For 1997–1999, he also joined the PRESTO project, Japan Science and Technology Corp. (JST). His research interests include theoretical aspects of computation, VLSI computing structures for signal and image processing, multiple-valued logic, and biomolecular computing. Dr. Aoki received the Outstanding Paper Award at the 1990, 2000, 2001 and 2006 IEEE International Symposia on Multiple-Valued Logic, the Outstanding Transactions Paper Award form the Institute of Electronics, Information and Communication Engineers (IEICE) of Japan in 1989 and 1997, the IEE Mountbatten Premium Award in 1999 IEEE International Symposium on Intelligent Signal Processing and Communication Systems, the IP Award at the Seventh LSI IP Design Award in 2005, the Best Paper Award at the 14th Workshop on Synthesis and System Integration of Mixed Information Technologies, and the Best Paper Award at the Workshop on Cryptographic Hardware and Embedded Systems 2014 (CHES 2014).