

# A Discrete Logarithm Number System for Integer Arithmetic Modulo $2^k$ : Algorithms and Lookup Structures

Alexandru Fit-Florea, Lun Li, Mitchell A. Thornton, *Senior Member, IEEE*, and David W. Matula, *Member, IEEE*

**Abstract**—We present a  $k$ -bit encoding of the  $k$ -bit binary integers based on a discrete logarithm representation. The representation supports a discrete logarithm number system (DLS) that allows integer multiplication to be reduced to addition and integer exponentiation to be reduced to multiplication. We introduce right-to-left bit serial conversion, deconversion, and unified conversion/deconversion algorithms between binary and DLS. The conversion algorithms utilize  $O(k)$  additions, do not require the use of a multiplier, and are applicable at least up to 128-bit integers. We illustrate the use of the representation in determining a novel and efficient integer power modulo  $2^k$  operation  $|x^y|_{2^k}$  and compare hardware performance with a current state-of-the-art method. Furthermore, we describe properties of the conversion mappings that allow compact table lookup structures to be employed for direct conversion to and deconversion from the DLS encoding. Our lookup architecture allows 16-bit conversion and deconversion mappings to be realized with table sizes of order 2-8 Kbytes, which is up to a  $64\times$  size reduction of the 128 Kbytes of an arbitrary 16-bits-in, 16-bits-out function table. Performance and area results that demonstrate the effectiveness of the table lookup architecture are given. The lookup methodology extends to other 16-bit integer functions such as multiplicative inverse and squaring operations.

**Index Terms**—High-speed arithmetic, arithmetic and logic units, computer arithmetic, discrete logarithm, number encodings, conversions, bit serial, integer power, table lookup.

## 1 INTRODUCTION

THIS paper's goal is to present the foundation for a discrete log representation and encoding of the integers with efficient conversion between standard binary radix bit string integer representation and bit string encoding of the discrete log representation of each integer. Our bit string encoding of the discrete-log integer representation employs  $k$  bits to represent the integers  $[0, 2^k - 1]$  in a scalable manner for all  $k$ . The representation employs reduction modulo  $2^k$  of a three-term product introduced by Benschop. In [1], Benschop showed that any  $k$ -bit integer  $x = b_{k-1}b_{k-2}\dots b_0$  can be represented by an exponent triple  $(s, p, e)$  satisfying the factored expression  $x = |(-1)^s 2^p 3^e|_{2^k}$ , where  $|\cdot|_{2^k}$  denotes the operation of reduction to the standard residue modulo  $2^k$  [9];  $s \in \{0, 1\}$ ; and  $p$  and  $e$  can be uniquely determined by upper bounds determined by  $k$ . Note that this representation allows for integer multiplication and powering to be executed more efficiently—much like in the case of real valued logarithms. In order to take advantage of this representation, efficient

methods to convert integers to and from the exponent triple are required. In the following, we discuss the mathematics involved and provide a number of algorithms for achieving this goal. We use the term discrete logarithm number system (DLS) to denote the representation of any integer  $x \in [0, 2^k - 1]$  by a triple  $(s, p, e)$ , and use the term DLS bit string for the  $k$ -bit encoding of the triple. Access to the separate exponents  $s, p$ , and  $e$  of the triple is useful for ALU design as with the separate processing of sign, exponent, and significant factors of a floating-point factorization  $v = (-1)^s 2^p (1.b_1b_2\dots b_{n-1})$ . The integrated DLS bit string is most suitable for efficient storage and table lookup.

This paper extends our previous papers focusing on the following:

- *Conversion/deconversion.* How do we implement the binary-integer-to- $(s, p, e)$  triple conversion and  $(s, p, e)$ -to-binary-integer deconversion? Since the exponent  $p$  can be determined by a count of low-order zeros in the binary radix integer bit string, the question reduces to how do we determine the  $(s, e)$  pair for an odd integer  $q$  such that  $q = |(-1)^s 3^e|_{2^k}$  in an efficient, reversible, and scalable manner.
- *Encoding.* How do we encode the triple  $(s, p, e)$  into a bit string with an appropriate integer range and convenient scalability for variable  $k$ -bit word sizes?

Our method can be summarized with regard to three distinct representations of integers by bit strings in a scalable manner parameterized by increasing  $k$ , where  $k$  denotes the  $k$ -bit integers, i.e., the set  $\{0, 1, 2, \dots, 2^k - 1\}$ :

• A. Fit-Florea is with Advanced Micro Devices (AMD), One AMD Place, P.O. Box 3453, MS 363, Sunnyvale, CA 94088-3453. E-mail: Alexandru.Fit-Florea@amd.com.

• L. Li is with Texas Instruments, MS 8736, 12500 TI Blvd., Dallas, TX 75243. E-mail: lun-li@ti.com.

• M.A. Thornton and D.W. Matula are with the Department of Computer Science and Engineering, Southern Methodist University, Dallas, TX 75205. E-mail: {mitch, matula}@engr.smu.edu.

Manuscript received 22 July 2007; revised 14 Feb. 2008; accepted 18 Sept. 2008; published online 24 Oct. 2008.

Recommended for acceptance by P. Kornerup, P. Montuschi, J.-M. Muller, and E. Schwarz.

For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-2007-07-0363. Digital Object Identifier no. 10.1109/TC.2008.204.

1. Representation A. Standard binary radix  $k$ -bit strings  $x = b_{k-1}b_{k-2} \dots b_0$ .
2. Representation B. The separable exponent triple binary integer bit strings  $(s, p, e)$ , where  $s \in \{0, 1\}$ , and  $p$  and  $e$  are determined uniquely in terms of  $k$  with  $x = |(-1)^s 2^p 3^e|_{2^k}$ .
3. Representation C. Our DLS  $k$ -bit string  $a_{k-1}a_{k-2} \dots a_0$  having a value of  $v(a_{k-1}a_{k-2} \dots a_0) = x$ , where parsing of the string yields unique integers  $s$ ,  $p$ , and  $e$  such that  $x = |(-1)^s 2^p 3^e|_{2^k}$ .

In Section 2, we provide the foundation for the encoding mapping between the triples  $(s, p, e)$  of Representation B and the  $k$ -bit DLS strings of Representation C that were introduced only by an example in [8]. Our DLS encoding employs self-determined variable length bit fields for encodings of parameters  $p$  and  $e$  within a  $k$ -bit word, in contrast to floating-point encodings, which employ fixed length fields for sign, exponent, and significand terms. A fundamental property of our  $k$ -bit encoding shown in Section 2 is that it is one-to-one over the integers  $[0, 2^k - 1]$  and constructed so that DLS strings satisfy the computationally useful inheritance property. This property provides that the  $j$ th bit ( $a_{j-1}$ ) of the DLS string  $a_{k-1} a_{k-2} \dots a_0$  depends only on the  $j$  low-order bits  $b_{j-1} b_{j-2} \dots b_0$  of the binary integer representation for all  $0 \leq j \leq k - 1$ . The inheritance property provides the theoretical foundation for bit serial algorithms. The reverse mapping from the DLS string to the  $(s, p, e)$  triple is accomplished by parsing. The parsing process first identifies parameter  $p$  and then the pair  $(s, e)$ .

In Section 3, we focus on the conversion/deconversion mappings between the binary radix bit strings of Representation A and the exponent triple  $(s, p, e)$  of Representation B. Efficient scalable iterative solutions of the integer-to-discrete log conversion and deconversion questions were presented at the algorithmic level in [2], [3], [4], and [16], and with hardware implementation in [6] and [17]. The algorithm described in [2] uses binary arithmetic with 3 as the logarithmic base and has a critical path containing one modulo  $2^k$  multiplication operation for each of its  $k$  iterations. Extensions of the algorithm to other logarithmic bases and computations using digits in a higher radix  $2^r$  are also described. The algorithm in [2] was improved in [3] by replacing  $k$  modulo  $2^k$  multiplication operations with  $k$  table lookup-determined shift-and-add modulo  $2^k$  operations. The algorithm described in [3] is well suited for implementation in special purpose hardware as also is a digit serial algorithm for deconversion modulo  $2^k$  introduced in [4]. Our new contribution in Section 3 is a unified conversion/deconversion algorithm extending results from [3] and [4]. The extension provides integration, potentially reducing hardware area requirements by almost one half.

As an internal ALU application of DLS representation, in Section 4, we describe and analyze the performance of a novel bit serial algorithm for the integer power operation  $|x^y|_{2^k}$  using the DLS triple as a catalyst. We extend the bit serial powering operation introduced in [7] by employing the unified conversion/deconversion from Section 3. Specifically, the design shows how the data path can be shared substantially reducing the total area for the whole

circuit. While the algorithms in [3] and [4] are scalable with increasing  $k$ , alternative compressed direct table lookup methods can be employed for sufficiently small values of  $k$ , such as values less than or equal to 16. In Section 5, we investigate a novel hierarchical table lookup architecture for direct binary-DLS bit string conversion traversing directly between Representations A and C. As mentioned in [8], the one-to-one hereditary property of DLS bit string encoding provides for table compression similar to that obtained for determining the multiplicative inverse in [5]. This hierarchical direct table lookup procedure demonstrates further options for table-assisted computation in optimizing an ALU design. Section 6 provides our conclusions highlighting the features and selective advantages of discrete log representation and encoding of the integers.

## 2 THE INHERITANCE PROPERTY AND DLS ENCODING

The binary radix integer bit string given by  $x = b_{n-1} b_{n-2} \dots b_0$  for  $0 \leq x \leq 2^n - 1$  implicitly denotes the radix polynomial  $x = \sum_{i=0}^{n-1} b_i 2^i$  with  $b_i \in \{0, 1\}$  for  $0 \leq i \leq n - 1$ . Partitioning the bit string yields

$$x = (b_{n-1}b_{n-2} \dots b_k) \times 2^k + b_{k-1}b_{k-2} \dots b_0$$

for any  $1 \leq k \leq n - 1$ . This means reduction modulo  $2^k$  is obtained simply by truncating the leading portion of the bit string. While straightforward, this reduction property as summarized in the following is a fundamental feature of radix representation.

**Observation 1.** *Given  $x = b_{n-1}b_{n-2} \dots b_0$ , then for  $1 \leq k \leq n - 1$ ,*

$$|x|_{2^k} = |b_{n-1}b_{n-2} \dots b_0|_{2^k} = b_{k-1}b_{k-2} \dots b_0.$$

The inheritance principle introduced in [5] applies to many integer operations and functions and is formally defined in terms of modular reduction as follows:

*Inheritance principle.* The integer operation  $z = x \otimes y$  has the inheritance property and is termed a hereditary operation if for all nonnegative integers  $x$  and  $y$ ,

$$|z|_{2^k} = ||x|_{2^k} \otimes |y|_{2^k}|_{2^k} \quad \text{for all } k \geq 1.$$

The integer function or unary operation  $z = f(x)$  has the inheritance property and is termed a hereditary function if for all nonnegative integers  $x$ ,

$$|z|_{2^k} = |f(|x|_{2^k})|_{2^k} \quad \text{for all } k \geq 1.$$

For example, integer addition and multiplication clearly are operations satisfying the inheritance property. The function  $x^2$  and more general  $x^y$  for any fixed  $y$  are hereditary functions.

In view of Observation 1, the inheritance principle may be interpreted as stating for hereditary operations and functions that the low-order  $k$  bits of the input arguments determine the low-order  $k$  bits of the output for all  $k \geq 1$ . With this interpretation, the inheritance principle is seen to be the basis for right-to-left bit serial algorithms such as grade school carry ripple addition.

**Observation 2.** Let  $z = f(x)$  be an integer valued hereditary function with input  $x = b_{n-1}b_{n-2}\dots b_0$  and output  $z = a_{n-1}a_{n-2}\dots a_0$ , then the  $k$ th bit of the output ( $a_{k-1}$ ) depends only on the low-order  $k$  bits of the input  $b_{k-1}b_{k-2}\dots b_0$ .

It follows from Observation 2 that an arbitrary hereditary function can be represented by a binary tree with edges labeled by the input bits and output bits at the vertices, with bit  $a_k$  at depth  $k+1$ . This lookup tree structure was introduced in [5] with specific regards to the modular multiplicative inverse function and table lookup size compression.

Our purpose in the rest of this section is to demonstrate the construction of a  $k$ -bit DLS encoding of the triple  $(s, p, e)$ , which provides a one-to-one hereditary function from binary integer radix representation directly to the DLS bit string.

To obtain the DLS encoding, first recall that a positive integer  $x$  has a unique factorization into odd and even terms  $x = 2^p q$  with  $q = b_k b_{k-1} \dots b_1 1$  being an odd integer. It is straightforward to show for  $k \geq 3$  that the  $2^{k-2}$  members of the sequence  $3^0, 3^1, 3^2, \dots, 3^{2^{k-2}-1}$  reduce modulo  $2^k$  to a sequence of distinct odd numbers covering half the odd numbers in  $[1, 2^k - 1]$ . The complementary values  $|(-1)^s 3^e|_{2^k}$  for  $s = 1, 0 \leq e \leq 2^{k-2} - 1$  cover the other half of the odd numbers. Furthermore,  $|3^{2^{k-2}}|_{2^k} = 1$  for all  $k \geq 3$  (but this does not hold for  $k = 2$ ). Thus, the sequence cycles with period  $2^{k-2}$  for all  $k \geq 3$ . For example, for  $k = 5$ , the reduced sequence is 1, 3, 9, 27, 17, 19, 25, 11, and the complementary sequence is 31, 29, 23, 5, 15, 13, 7, 21. Thus, every odd  $k$ -bit integer is uniquely represented by the exponent pair  $(s, e)$  in the DLS representation.

A  $k-1$  bit encoding of the odd integers  $q = b_k b_{k-1} \dots b_1 1$  by pairs  $(s, e)$  can be formed as follows: The powers  $|3^e|_{2^k}$  for  $0 \leq e \leq 2^{k-2} - 1$  all have bit  $b_2 = 0$ , and their complements  $|(-1)3^e|_{2^k}$  all have  $b_2 = 1$ . Thus, let  $s = b_2$  with  $e = e_{k-3}e_{k-4}\dots e_0$  determined, satisfying  $|3^e|_{2^k} = q = b_{k-1}b_{k-2}\dots b_3 0 b_1 1$  for  $b_2 = 0$  and  $|(-1)3^e|_{2^k} = 2^k - q = b'_{k-1}b'_{k-2}\dots b'_3 0 b'_1 1$  for  $b_2 = 1$ .

**Lemma 3.** For any odd integer  $q = b_{k-1}b_{k-2}\dots b_1 1$  with  $k \geq 3$ , there is a unique exponent  $e = e_{k-3}e_{k-4}\dots e_0$  and sign bit  $s = b_2$  satisfying  $q = |(-1)^s 3^e|_{2^k}$ . Furthermore, the bits  $b_2, b_1$ , and  $e_0$  are related by  $e_0 = b_2 \text{ xor } b_1$  and  $s = b_2 = e_0 \text{ xor } b_1$ .

*Sketch of proof.* The equation for  $q$  follows from the preceding discussion. Noting that odd powers of 3 have  $e_0 = 0$  and  $b_1 = 1$  and that  $s = b_2$ , the relations between  $b_2, b_1$ , and  $e_0$  are obtained.

**Definition.** For  $k \geq 3$ , the DLS encoding  $DLS(s, e) = a_{k-1}a_{k-2}\dots a_1 1$  determined given  $s$  and  $e = e_{k-3}e_{k-4}\dots e_0$  for the odd integer  $q = |(-1)^s 3^e|_{2^k}$  is the bit string  $DLS(s, e) = e_{k-3}e_{k-4}\dots e_0 b_1 1$  with  $b_1 = e_0 \text{ xor } s$ . For  $k = 2$ , the sign bit is assumed to be  $s = 0$  with  $DLS(s, e) = e_0 1$ , equivalently  $DLS(s, e) = b_1 1$ . For  $k = 1$ , corresponding to  $q = 1$ , the DLS encoding is the unit bit. The DLS encoding of an even integer  $x = q2^p = |(-1)^s 2^p 3^e|_{2^k}$  for  $p \neq 0$  is obtained by appending  $p$  low-order zeros to the DLS encoding of  $q = |(-1)^s 3^e|_{2^{k-p}}$ .

**Theorem 4.** The DLS encoding is a one-to-one hereditary function for all odd integers.

*Sketch of proof.* For any  $3 \leq j \leq k$ , note that  $|3^{2^m}|_{2^j} = 1$  for all  $m \geq j-2$ , so that  $k-j$  leading bits of  $e$  do not affect the determination of  $|q|_{2^j} = |(-1)^s 3^{2^{k-j}}|_{2^j}$ . For the transition cases  $k=2$  and  $k=1$ , the result can be verified by enumeration.

The extension of Theorem 4 to even integers is immediate noting that the DLS encoding includes the same  $(p+2)$  low-order bits as the binary representation of  $x = 2^p q$ .

The one-to-one mapping between 5-bit discrete log numbers comprising a 5-bit DLS encoding and standard 5-bit binary radix representation is given in Table 1. The DLS bit string is partitioned as follows to determine the three exponents  $p, e$ , and  $s$ .

Consider the line in the table for DLS string  $a_4 a_3 a_2 a_1 a_0 = 10110$  (highlighted in Table 1). The parsing begins from the right-hand side, determining the variable length field identifying  $2^p = a_1 a_0 = 10_2$  by counting zeros until the first unit bit is encountered. The 2-bit field “unary” encoding of  $p$  determines  $p = 1$ . The next bit is a separation bit providing the logical value  $s \oplus e_0$  used to determine  $s$  after  $e$  is determined. The remaining leading bits are the  $5 - (p+2)$  bits of the exponent  $0 \leq e \leq 2^{5-(p+2)} - 1$  sufficient to determine the odd factor  $q = |(-1)^s 3^e|_{2^k}$ . Thus,  $x = q \times 2^p$  is the integer represented with  $0 \leq x \leq 2^5 - 1$  uniquely determined. In this example,  $e = 10_2 = 2_{10}$ , and then  $s = 1$  is determined from  $e_0 = 0$  and  $s \oplus e_0 = 1$ . Finally,  $|(-1)^1 2^1 3^2|_{32} = |-18|_{32} = 14$  or  $b_4 b_3 b_2 b_1 b_0 = 01110$  is obtained. Note that the low-order  $p+2$  bits are identical in both DLS and binary integer encodings.

The conversion/deconversion for odd integers in Table 1 can be visualized by the lookup trees illustrated in Figs. 1a and 1b, respectively. Navigation in Fig. 1a for binary-to-DLS conversion occurs by reading down with edge direction determined by the 5-bit odd integer string read right-to-left. The DLS output string  $a_4 a_3 a_2 a_1 a_0$  is obtained (right to left) from the bits extracted from the vertices. For example, determining the DLS of the integer  $13 = b_4 b_3 b_2 b_1 b_0 = 01101$  is achieved by traversing the tree in Fig. 1a one bit at a time, starting with the least significant bit (LSB)  $b_0 = 1$  traversing to the right from the root of the tree, determining  $a_0 = 1$ , then  $b_1 = 0$  dictates traversing to the left determining  $a_1 = 0$ . Bits  $b_2 = 1, b_3 = 1$ , and  $b_4 = 0$  continue right, right, then left, determining  $a_2 = 1, a_3 = 0$ , and  $a_4 = 1$ . A similar traversal of the lookup tree exists for Fig. 1b. The lookup tree structure illustrates the use of the hereditary property of these conversion operations.

### 3 DLS CONVERSION AND DECONVERSION ALGORITHMS

Exponent  $p$  for an even number can be determined by counting the low-order zeros in the binary radix integer bit string. Without loss of generality, we focus on odd numbers for the rest of this paper. Binary-to-discrete log conversion refers to determining the pair  $(s, e)$  given the  $k$ -bit odd integer  $q$ , and deconversion refers to determining  $q$  given the pair  $(s, e)$ , where  $q, s, e$  satisfy  $q = |(-1)^s 3^e|_{2^k}$ . For

TABLE 1  
Conversion Table from the 5-Bit DLS to the 5-Bit Integers

Discrete Log Number System (DLS) Encoding	Partitioned DLS Bit Strings			Integer Value	Standard Binary	Integer Parity
	$e$	$e_0 \text{ xor } s$	$2^p$	$\left (-1)^s 2^p 3^e\right _{32}$		
00001	000	<b>0</b>	1	1	00001	Odd
00011	000	<b>1</b>	1	31	11111	
00101	001	<b>0</b>	1	29	11101	
00111	001	<b>1</b>	1	3	00011	
01001	010	<b>0</b>	1	9	01001	
01011	010	<b>1</b>	1	23	10111	
01101	011	<b>0</b>	1	5	00101	
01111	011	<b>1</b>	1	27	11011	
10001	100	<b>0</b>	1	17	10001	
10011	100	<b>1</b>	1	15	01111	
10101	101	<b>0</b>	1	13	01101	
10111	101	<b>1</b>	1	19	10011	
11001	110	<b>0</b>	1	25	11001	
11011	110	<b>1</b>	1	7	00111	
11101	111	<b>0</b>	1	21	10101	
11111	111	<b>1</b>	1	11	01011	
00010	00	<b>0</b>	10	2	00010	Singly Even
00110	00	<b>1</b>	10	30	11110	
01010	01	<b>0</b>	10	26	11010	
01110	01	<b>1</b>	10	6	00110	
10010	10	<b>0</b>	10	18	10010	
<b>10110</b>	<b>10</b>	<b>1</b>	<b>10</b>	<b>14</b>	<b>01110</b>	
11010	11	<b>0</b>	10	10	01010	
11110	11	<b>1</b>	10	22	10110	
00100	0	<b>0</b>	100	4	00100	Doubly Even
01100	0	<b>1</b>	100	28	11100	
10100	1	<b>0</b>	100	20	10100	
11100	1	<b>1</b>	100	12	01100	
01000		<b>0</b>	1000	8	01000	Triply Even
11000		<b>1</b>	1000	24	11000	
10000			10000	16	10000	Quad. Even
00000			00000	0	00000	Zero

conversion,  $s$  is determined by conditional complementation to obtain a normalized  $q$ . Without loss of generality, assume  $s = b_2 = 0$ , so that  $q$  is congruent to 1 or 3 (mod 8). This reduces the conversion operation to the determination of the discrete log  $e = \text{dlg}(q)$ , with  $0 \leq e \leq 2^{k-2} - 1$  and  $q = 1, 3 \pmod{8}$ . The deconversion problem reduces to evaluating the exponential residue operation determining  $q$ , where  $q = |3^e|_{2^k}$ . For completeness, we review algorithms from [3] and [4] demonstrating that both the exponential residue operation (determining  $q$  given  $e$ ) and the discrete log operation (determining  $e$  given  $q$ ) can be performed by a series of less than  $k$  table-assisted shift-and-add operations employing *exponent recoding*.

We conclude this section by emphasizing the algorithmic similarities between additive conversion and deconversion and introduce a unified conversion/deconversion algorithm. This has the attractive property of reducing the hardware area required due to sharing a common data path

as compared to independent conversion and deconversion algorithm implementations in circuitry.

### 3.1 Additive Exponentiation Modulo $2^k$

$|3^e|_{2^k}$  can be computed using the square-and-multiply method [12]. This entails computing  $|3^2|_{2^k}, |3^4|_{2^k}, \dots, |3^{2^k}|_{2^k}$ , by successive squaring. We observe that similar methods lead to the correct result when the exponent  $e$  is recoded as a sum of elements  $e = |\sum \alpha_i|_{2^{k-2}}$  [4]. In this case,  $|3^e|_{2^k}$  can be computed as  $|3^e|_{2^k} = |3^{\sum \alpha_i}|_{2^k}$ . Of course, this presents an advantage if the  $\alpha_i$  and/or corresponding powers  $\{3^{\alpha_i}\}$  are precomputed and available by an exponent table lookup. In [4], it is shown that any exponent  $e$  can be expressed as a sum of  $\text{dlg}(2^i + 1)$ 's termed the *two-ones discrete logs*. Since  $3^{\text{dlg}(2^i + 1)} = 2^i + 1$ , it follows that the corresponding multiplications can be performed as a series of shift-and-add

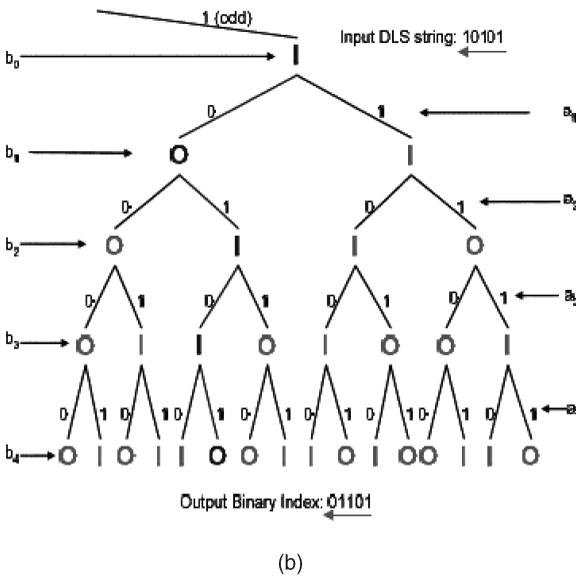
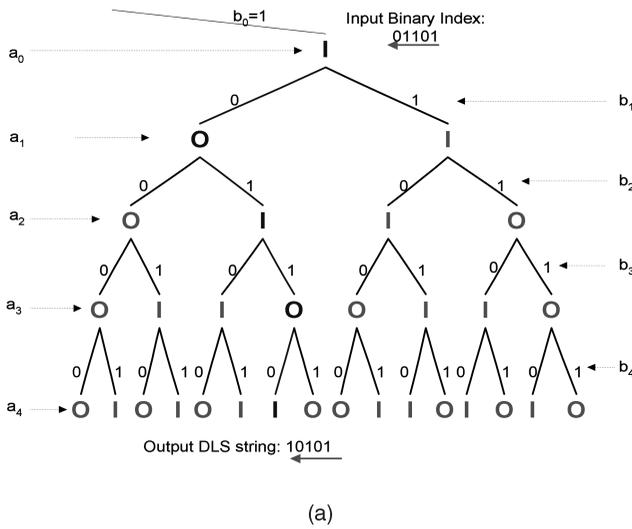


Fig. 1. (a) Five-bit lookup tree for odd integer binary to DLS encoding. (b) Five-bit lookup tree for DLS encoding to odd integer binary.

operations. This works if the two-ones  $\text{dlg}$ 's are precomputed and stored in a table (as shown in Table 2) of just  $k$  entries.

Algorithm 1 [4] determines the unique set of  $\text{dlg}(2^i + 1)$ 's whose sum modulo  $2^{k-2}$  is equal to  $e$ . It thus allows efficient conversion from DLS to binary. In the algorithmic description that follows, index notation is used for the corresponding bit of the standard binary representation. As an example, if a value  $x$  is formed as the bit string  $x_{n-1}x_{n-2}\dots x_2x_1x_0$ , the notation  $x_i$  refers to the bit with subscript  $i$ .

#### Algorithm 1 DLS-to-Binary Deconversion Algorithm (EXP)

**Stimulus:**  $k$ ;  $e = e_{k-3}e_{k-4}\dots e_2e_1e_0$

**Response:**  $|3^e|_{2^k}$

**Method:**

- L1: if  $e_0 = 1$ , then  $y := 3$ ;  $s := e - 1$  /\* L1-L3: Initialize arguments and result \*/  
 L2: else  $z := 1$ ;  $h := e$   
 L3: end

TABLE 2  
Two-Ones Discrete Log Table for  $k = 8$

$i$	$2^i + 1$		$\text{dlg}(2^i + 1)$		check
	Bin.	Dec.	Bin.	Dec.	
0	0000 0001	1	0000 0000	0	$ 3^0 _{256} = 1$
1	0000 0011	3	0000 0001	1	$ 3^1 _{256} = 3$
2	0000 0101	5	N/A	N/A	N/A
3	0000 1001	9	0000 0010	2	$ 3^2 _{256} = 9$
4	0001 0001	17	1011 0100	7604	$ 3^{7604} _{256} = 17$
5	0010 0001	33	0010 1000	15912	$ 3^{15912} _{256} = 33$
6	0100 0001	65	0101 0000	10064	$ 3^{10064} _{256} = 65$
7	1000 0001	129	1010 0000	15008	$ 3^{15008} _{256} = 129$

L4: for  $i := 1$  to  $k - 3$  do

/\* L4: Loop over bits of  $h$ , index 1 to  $k - 3$  \*/

L5: if  $h_i = 1$ , then /\* L5: Conditional on  $i$ th bit of  $h$  \*/

L6:  $z := |z \times (2^{i+2} + 1)|_{2^k}$

/\* L6: Update response  $z$  by shift and add \*/

L7:  $h := h - \text{dlg}(2^{i+2} + 1)$

/\* L7: Update argument  $h$  to reflect update to  $z$  \*/

L8: end if

L9: end for loop

L10: Result:  $(z)$  /\* L10: Return binary result  $z$  \*/

Note that lines L1-L3 correspond to initialization. The product  $z$  is set to either 1 or 3 (corresponding to  $e_0 = 1$  or  $e_0 = 0$ ). The working variable exponent  $h$  is always set in such a way that  $z$  corresponds, for each iteration, to 3 raised to the exponent  $(e - h)$  and the least significant  $i$  bits of  $h$  are all 0s. The algorithmic step of lines L4-L8 represents updating  $h$  by subtracting  $\text{dlg}(2^{i+2} + 1)$ , which simply represents the exponent of 3 that reduces to  $2^{i+2} + 1$ . This is followed by updating the product  $z$  to reflect the changes in exponent,  $z := |z \times (2^{i+2} + 1)|_{2^k}$ . Eventually, after  $(k - 2)$  steps,  $h$  becomes 0 and the "product"  $z$  corresponds to  $|3^{e-0}|_{2^k} = |3^e|_{2^k}$ . The values  $\text{dlg}(2^{i+2} + 1)$  can be stored in a lookup table and this method is practical for large  $k = 64, 128, \dots$ , since the table has only  $k$  entries.

### 3.2 Additive-Based Discrete Logarithm Modulo $2^k$

Computing the discrete logarithm for certain  $k$ -bit odd integers  $x$  can be accomplished using a method [3] that is essentially the dual of the exponentiation method of Section 3.1. The key idea is to express  $x$ , if possible, as a product of two-ones residues:  $x = |\prod (2^i + 1)|_{2^k}$  for selected  $i$ 's. Once this is done, the discrete logarithm can be computed as the corresponding sum:  $\text{dlg}(x) = \text{dlg}(\prod (2^i + 1)) = |\sum \text{dlg}(2^i + 1)|_{2^{k-2}}$  [3]. The solution involves identifying the cases when  $x$  can be expressed as such a product and finding the corresponding unique set of two-ones residues. It is

shown in [3] that  $x$  can be expressed as a two-ones residue product as long as  $x$  is congruent with 1 or 3 modulo 8. Note that for the remaining odd residues, corresponding to  $x$  congruent with 5 or 7 modulo 8, their additive inverses  $| -x |_{2^k}$  are congruent with 1 or 3 modulo 8. The method in [3] identifies the set of two-ones residues, and thus, it is the core of a digit serial conversion method from binary to DLS.

**Algorithm 2** Binary to DLS Conversion Algorithm (DLG)

**Stimulus:**  $k, x = x_{k-1}x_{k-2} \dots x_2x_1x_0$  with  $x_0 = 1$  (odd values)

**Response:** discrete log of  $x$ , expressed as an  $(s, e)$  pair, where  $x = |(-1)^s 3^e|_{2^k}$ .

**Method:**

```

L1:  if  $b_2 = 1$ , then  $x := 2^k - x$ ;
    /* L1: Get 2's complement of  $x$  if  $b_2 = 1$  */
L2:  end if
L3:   $t := 1$ ;  $e := 0$ ;  $s := b_2$ 
    /* L3: Initialize arguments  $t, e,$  and  $s$  */
L4:  for  $i := 1, 3$  to  $k - 1$  do
    /* L4: Loop over bit indices 1 to  $k - 1$  skipping  $i = 2$  */
L5:    if  $x_i = t_i$ , then
    /* L5: Conditional on equivalence of bits  $x_i$  and  $t_i$  */
L6:       $t = t \times (2^i + 1)$ ;
    /* L6: Conditionally update  $t$  with shift and add */
L7:       $e := e + \text{dlg}(2^i + 1)$ 
    /* L7: Conditionally update  $e$  with lookup from dl $g$  table */
L8:    end if
L9:  end for loop
L10: Result:  $(s, e)$  /* L10: Return result as  $(s, e)$  pair */

```

The initialization stage is performed in lines L1-L3. If  $x$  is complemented, then  $b_2 = 1$ . The second stage contains the main iteration step and is represented by lines L4-L9, where both  $p$  and the exponent  $e$  are updated.  $p$  is conceptually updated as  $t = t \times (2^i + 1)$ , while the exponent  $e$  is updated by adding the corresponding values  $\text{dlg}(2^i + 1)$ , looked up from a table. The final result is returned in line L10 as the sign  $s$  and the exponent  $e$  pair. The updating of  $e$  and in lines L6 and L7 can be performed concurrently. As can be seen by inspection of Algorithm 2, the time complexity is essentially  $k$  dependent shift-and-add modulo  $2^k$  operations.

### 3.3 Unified Conversion/Deconversion Algorithm

Similarities between the additive versions of the deconversion (Algorithm 1) and conversion (Algorithm 2) algorithms presented in Sections 3.1 and 3.2 are described here. Since there are minimal differences between the two algorithms, they are very suitable for hardware reuse and a unified algorithm is developed.

While conceptually one operation is the inverse of the other, they can be executed on the same data path. An intuitive explanation as to why this is feasible is presented here.

Algorithm 2 computes the discrete log of  $x$ . In order to do this,  $e$ , the discrete log of  $x$ , is updated one digit at a time. Concurrently,  $t$  is updated to eventually become  $|x \times x^{-1}| = 1$ . The way  $t$  and  $e$  are updated is strongly related in the sense that  $t$  is multiplied with  $(2^i + 1)$  while

$e$  is adjusted by  $\text{dlg}(2^i + 1)$ , its discrete log. Eventually,  $e$  represents the discrete log of  $\pm x$ .

Algorithm 1 starts with product  $z = 1$  and updates it by multiplying with selected two-ones residues,  $(2^i + 1)$ 's. Concurrently,  $s$ , the exponent, is correspondingly adjusted by subtracting  $\text{dlg}(2^i + 1)$ . This way, the multiplications by  $(2^i + 1)$  are counted off from the exponent. Eventually, the exponent becomes zero valued and the product becomes  $z = |3^e|_{2^k}$ .

In Algorithm 1,  $s$  is updated the same manner as  $e$  of Algorithm 2 but with a sign reversal. It is already shown that  $|\text{dlg}(z)|_{2^{k-2}} = | -\text{dlg}(z^{-1}) |_{2^{k-2}}$ . Computing  $z = |3^e|_{2^k}$  as  $z = |3^{-(e)}|_{2^k}$  allows changing the update of  $t$  by  $+\text{dlg}(2^i + 1)$  as opposed to  $(-\text{dlg}(2^i + 1))$  before and inside the core loop of Algorithm 1. Due to this switching, the algorithm would require some changes in the initialization part, but it would still produce  $s = \text{dlg}(z)$  at the end.

As emphasized above, the two algorithms can have a common core—the iteration loop, and the only differences are the initialization steps and the result returned by the algorithm. Last, but not the least, the table lookup uses the same table and the same entry for equal values of the loop counter. This allows us to introduce the Unified Deconversion-Conversion Algorithm.

**Algorithm 3** Unified Deconversion—Conversion Algorithm

**Stimulus:**

operation  $\otimes$ : either conversion or deconversion  
 $k, x = x_{k-1}x_{k-2} \dots x_2x_1x_0$ : either residue or exponent (with  $x_0 = 1$  for EXP),

**Response:**

discrete log of  $x$ , when  $\otimes$  is conversion  
 or  $|3^e|_{2^k}$ , when  $\otimes$  is deconversion

**Method:**

```

L1:  if  $\otimes$  is conversion, then
    /* L1: Conditional on conversion or deconversion */
L2:    if  $b_2 = 1$ , then  $x := 2^k - x$ ;
    /* L2: Get 2's complement of  $x$  if  $b_2 = 1$  */
L3:    end if
L4:     $z := 1$ ;  $e := 0$ ;  $s := b_2$ 
    /* L4: Initialize arguments  $t, e,$  and  $s$  */
L5:  else /* L5-L9: Initialization for deconversion */
L6:    if  $x_0 = 1$ , then  $z := 3$ ;  $e := 2^{(k-2)} - x + 1$ 
    /* L6: Conditional on LSB of  $x$  */
L7:    else  $z := 1$ ;  $e = 2^{(k-2)} - x$ 
    /* L7: Initialize arguments  $p$  and  $e$  */
L8:  end if
L9:  end if
L10: for  $i := 1$  to  $k - 3$  do
    /* L10: Loop over bit indices 1 to  $k - 3$  */
L11:  if  $e_i = 1$ , then
    /* L11: Conditional on  $i$ th bit of  $e$  */
L12:     $z := |z \times (2^{i+2} + 1)|_{2^k}$ 
    /* L12: Update  $p$  with multibit left shift */
L13:     $e := e + \text{dlg}(2^{(i+2)} + 1)$ 
    /* L13: Update  $e$  with lookup from dl $g$  table */
L14:  end if
L15: end for loop
L16: if  $\otimes$  is conversion, then
    /* L16: Conditional on conversion or deconversion */

```

```

L17:  Result: (z)      /* L17: Return binary result */
L18:  else
L19:  Result: (s, e)
      /* L19: Return discrete log result */
L20:  end if

```

As mentioned previously and also shown in [7], the DLS together with conversion/deconversion algorithms provide an efficient method for the integer powering modulo  $2^k$  operation. In Section 4, we present a DLS-based method for the integer powering operation that uses the conversion/deconversion algorithms above and an analysis of its hardware implementation.

## 4 INTEGER POWERING USING INTERMEDIATE DLS REPRESENTATION

Algorithms for computing the operation  $z = x^y$ , where  $y$  is a positive integer, have been the subject of considerable research. The binary square-and-multiply method determines  $x, x^2, x^4, x^8, \dots$  and processes the bits of  $y$  from right-to-left to multiply by the appropriate binary powers of  $x$  to determine  $x^y$ . This algorithm has been described in many popular texts [10], [11], [12]. Knuth [12] traces this “fast” algorithm back to al-Kashi in the 15th century.

We are interested in the particular case of  $z = |x^y|_{2^k}$ , where  $x$ ,  $y$ , and the result  $z$  are all nonnegative  $k$ -bit integers. For typical word sizes such as  $k = 8, 16, 32, 64, 128$ , and so forth, this integer valued powering operation is proposed to supplement the integer addition and multiplication operations. The squaring algorithm may be implemented in hardware with microcode and a fast multiplier much like the floating-point transcendental operations in the Pentium and Athlon processors.

For implementation in hardware, there is a need for a simpler algorithm that avoids the use of a large multiplier. There is a further need for a right-to-left digit serial algorithm that requires less time for lower precision operations when a family of precision levels is implemented in hardware.

In this section, we present a novel digit serial algorithm for evaluation of the integer power operation  $|x^y|_{2^k}$  that is based on Algorithm 3; hence, it does not require a multiplier. The algorithm employs both conversion and deconversion of  $x$  to and from DLS, and bit serial multiplication. The conversion of the input  $x$  to DLS is implemented with bit serial multiplication and the discrete log (converted) value provides the “recoded multiplier bits” with the exponent  $y$  being the multiplicand and bit serial deconversion of  $|x^y|_{2^k}$  provides the result  $z$ . In the following, we first describe the existing “Fast” binary squaring algorithm [10], [11], [12].

### 4.1 Existing “Fast” Binary Squaring Algorithm

The existing fast algorithm is based on the fact that  $y = \sum_{i=0}^{k-1} y_i 2^i$ , so that we can get the formula

$$z = |x^y|_{2^k} = \left| x^{\sum_{i=0}^{k-1} y_i 2^i} \right|_{2^k} = \left| x^{y_0 2^0} \times x^{y_1 2^1} \times \dots \times x^{y_{k-1} 2^{k-1}} \right|_{2^k}.$$

As an example,  $3^{10} = 3^{0 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3} = 3^2 \times 3^8$ . This method is implemented in Algorithm 4.

### Algorithm 4 Binary Squaring Powering ( $x^y$ )

**Stimulus:**  $k, x = x_{k-1}x_{k-2} \dots x_2x_11$ ,

$y = y_{k-1}y_{k-2} \dots y_2y_1y_0$

**Response:**  $z = |x^y|_{2^k}$ .

```

L1:  z := 1; h := x;      /* Initialize Response Value z
                          to 1 and Argument h to x */
L2:  for i := 0 to k - 1, do /* Loop over the number
                          of bits in the argument x */
L3:  if bit  $y_i = 1$ , then /* If the  $i$ th bit (denoted  $y_i$ ) of
                          the exponent  $y$  is 1 */
L4:  z := |z × h|2k /*, then update output value z to
                          contain the factor h */
L5:  end if
L6:  h := |h × h|2k /* Update argument h to hold
                          next higher power of x */
L7:  end for loop
L8:  Result: (z)
      /* Response z now holds the  $x^y$  modulo  $2^k$  value */

```

### 4.2 Proposed Feedback Shift Add (FSA) Algorithm

Any number can be converted to a triple  $(s, p, e)$ , where  $x = 2^p q$  with  $q$  being an odd integer. Thus

$$z = |x^y|_{2^k} = |((-1)^s 2^p 3^e)^y|_{2^k} = |(-1)^{sy} 2^{py} 3^{ey}|_{2^k}.$$

In the above formula,  $(-1)^{sy}$  determines the sign.  $2^{py}$  determines the number  $(py)$  of least significant zeros. For odd numbers, we need to calculate  $e \times y$  for term  $3^{ey}$ . Then, we can convert  $|(-1)^{sy} 3^{ey}|_{2^k}$  back to binary to obtain  $z$ .

Computing the  $y$ th power of operand  $x$  can be done in a serial fashion. That is, we start multiplication and decoding after we obtain the entire value of  $e$ . A better technique is a pipelined arrangement of the suboperations in which multiplication and decoding starts when the first bit of  $e$  is available. For every available bit of  $e$ , a bit of the intermediate product is generated followed by a bit of  $z$  being produced. This method is referred to as the pipelined algorithm and is described in the following algorithm.

### Algorithm 5 Additive Digit Serial Powering ( $x^y$ )

**Stimulus:**  $k, x = x_{k-1}x_{k-2} \dots x_2x_11$ ,

$y = y_{k-1}y_{k-2} \dots y_2y_1y_0$

**Response:**  $z = |x^y|_{2^k}$ .

#### Method

```

L1:  if  $b_2 = 1$ , then  $x := 2^k - x$ ;
      /* L1: Get 2's complement of  $x$  if  $b_2 = 1$  */
L2:  end if
L3:  t := 1; e := 0; z := 1; h := 0; g = e; s :=  $b_2$ ;
      /* L3: Initialize t, e, z, h, g, s */
L4:  if  $x_1 = t_1$ , then
      /* L4-L7: Update p (shift) and e (table) if  $x_1$  is  $p_1$  */
L5:  t =  $t \times (2^i + 1)$ ; e =  $e + \text{dlg}(3)$ 
L6:  z := |z × (2 $y_0$  + 1)|2k
      /* L6: Response z is updated if  $y_0 = 1$  */
L7:  end if
L8:  for i := 3 to k - 1 do
      /* L8: Loop over bit index values 3 to k - 1 */
L9:  if  $x_i = p_i$ , then

```

```

/* L9-L11: Update t (shift) and e (table) if x1 is p1 */
L10:  t = t × (2i + 1); e := e + dlg(2i + 1)
L11:  end if
L12:  g = 2 × g; /* L12-L13: Update accumulated
value in variable m */
L13:  m = m + g × ei-2
L14:  if mi-2 = 1, then /* L14-L16: Update h with
variable shift and add */
L15:  h = h × (2i-2 + 1)
L16:  end if
L17:  if hi = 1, then /* L17-L19: Conditionally
update z and h based on hi */
L18:  z := |z × (2i+2 + 1)|2k; h := h - dlg(2i+2 + 1)
L19:  end if
L20: end for loop

```

The initialization stage is performed in lines L1-L4. All the required initializations for both stages of the algorithm are performed here. The second stage (L5-L7) actually performs the computation for the next to the last LSB with index  $i = 1$ . The third stage contains the main iteration step and is represented by lines L8-L20. The third stage can be separated into three substages. Both  $t$  and the exponent  $e$  are updated (i.e., L9-L11), which generates 1 bit of exponent according to the DLG algorithm. The second substage (i.e., L12-L16) corresponds to the accumulator used to implement  $e \times y$ . The third stage updates  $z$  according to EXP algorithm (i.e., L17-L19). The final result is obtained at line L20. As can be seen by inspection of the algorithm, the time complexity is essentially  $k$  dependent shift-and-add modulo  $2^k$  operations.

### 4.3 Hardware Performance Evaluation for Integer Power Operation

In order to evaluate the effectiveness of our method as compared to the well-known “fast” squaring method, we described each method in Verilog RTL. To implement the “fast” squaring method described in Algorithm 4, a counter is employed to control the number of loops, and the values  $z$  and  $q$  are updated simultaneously. There are three major components in the implementation of the circuit described in Algorithm 5: a controller, a ROM lookup table, and a computation data path. The controller consists of a counter and state control block, Finite State Machine (FSM). The FSM will start and stop the counting procedure. The output of the counter, **count**, is used for purposes such as address generation for the ROM, index production for the bit checker and loop controller, and feedback to the FSM for state transition. The ROM is used as a lookup table for the  $\text{dl}_g(\tau)$  function. The major components in the data path are adders, shifters, and muxes. The muxes are used to control whether registers holding  $p$ ,  $e$ ,  $z$ , and  $q$  will be updated by the shifter and adder circuits. The modulo operation used in the description of the algorithms is handled by limiting the register size of  $p$ ,  $e$ ,  $z$ , and  $q$ . The width of the register containing  $p$ ,  $e$ ,  $z$ , and  $q$  is set to  $k$ . Thus, while updating  $p$ ,  $e$ ,  $z$ , and  $q$ , the result values may be longer than the specified size (causing overflow). This intentional overflow actually implements the modulo  $2^k$  operation. The two designs corresponding to Algorithms 4 and 5 are implemented and both are synthesized using the Synopsys toolset based on a

TABLE 3  
Comparison of Layout Result

Wordsize $k$	Period(ns)		core area( $\mu\text{m}^2$ )	
	ours	fast	ours	Fast
8	2.05	2.4	23386.4	8207.48
16	2.41	3.45	40306.7	26076.3
32	2.75	4.55	109135	79409.1
64	3.52	5.55	184725	302942
128	3.8	6.8	371366	1.26E+06

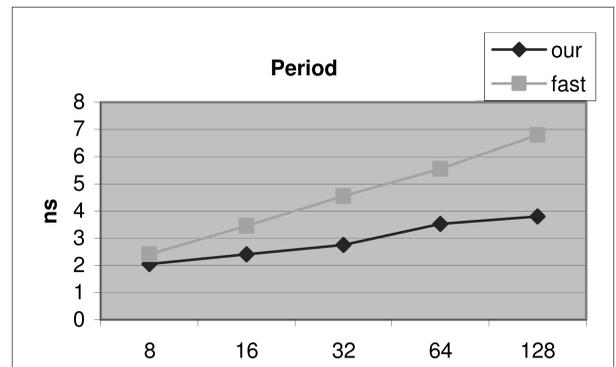


Fig. 2. Period trend of two algorithms.

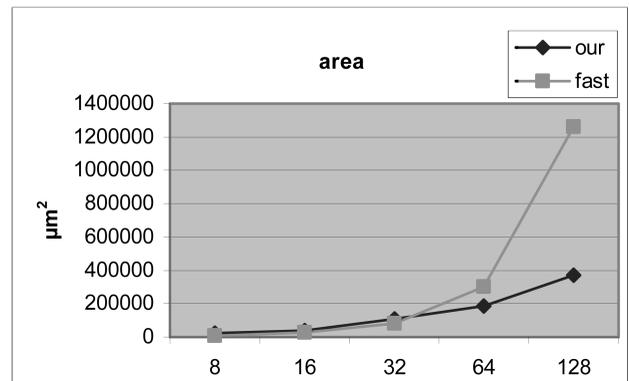


Fig. 3. Area trend of two algorithms.

standard cell library from Synopsys [13] and a standard cell library from Oklahoma State University [14]. Since the results from the two standard cell libraries were similar, we only list the result based on the standard cell library from Synopsys.

Table 3 compares the results of our Algorithm 5 with the existing fast algorithm (Algorithm 4) for different  $k$  values. The latency of both designs are  $k$  since they are all bit serial based. We also plot the trend of the two algorithms in Figs. 2 (period) and 3 (area). It is seen that for all  $k$  values, our algorithm is faster than the existing fast algorithm when each algorithm is synthesized with the standard cell library. Regarding area, our method requires more space for small word sizes but increases slowly compared with the existing fast algorithm. Thus, when  $k \geq 64$ , our algorithm requires less area. It should be noted that the area values reported here are only the net area required by the total cell area

since we did not route the resulting circuits, thus additional area and delay required by routing are not included.

As it is often the case, for relatively small values of the word size, a specific function can be looked up in a table as opposed to being computed using a functional hardware implementation. This would allow for speeding up computations at the expense of area. In Section 5, we present the complementary part of the functional algorithms in Section 3 by introducing DLS-related lookup structures and efficient compression methods.

## 5 LOOKUP STRUCTURE-BASED METHOD

Integer functions that are determined modulo  $2^k$  for  $k$ -bit word results generally have properties allowing much smaller tables for exhaustive storage of  $k$ -bits-in,  $k$ -bits-out function evaluation than corresponding real valued  $k$ -bit functions. Four properties of integer functions are identified, which, when used in combination, allow for significant reduction in size of lookup tables for integer functions adhering to these properties (for 16-bit arguments, the advantages for integer function lookup can be as large as 32-to-1 or even 64-to-1, allowing 5 or 6 more index bits for comparable table size):

1. *Inheritance principle.* Briefly, this principle states that the low-order  $k$  bits of the result depend only on the low-order  $k$  bits of the integer argument for all  $k$ . In practice, the inheritance principle for integer functions means that a  $k$ -bits-in,  $k$ -bits-out lookup table can be reduced from a generic  $k \times 2^k$  bits ROM table to a lookup tree structure of size  $2 \times 2^k$  bits. This reduces table size by a factor of  $k/2$  (e.g., reduction to 1/8 the size for 16-bit integers).
2. *One-to-one correspondence.* This property holds when distinct  $k$ -bit inputs have distinct  $k$ -bit outputs. This property holds for multiplicative inverse and the discrete log of odd integers and is extended to a discrete log encoding of all  $k$ -bit integers as illustrated in Section 2. With the inheritance principle, this property allows pre and postprocessing logic to reduce the table size by another half.
3. *Normalization (separating odd and even factors).* Employing a right-normalized binary integer representation,  $x = 2^p q$  (where  $q$  is the odd factor and  $2^p$  is the even-power factor), integer functions can often be determined in a separable fashion by applying table lookup to the argument's odd factor followed by function-specific postprocessing responsive to the even-power factor.
4. *Conditional complementation.* This property states that the result of the operation on the conditional 2's complement of the input is the conditional 2's complement of the output. Conditional complementation often applies only to selected bits of the odd factor of the normalized integer argument. When applicable, this allows one half or more further table size reduction.

To fully benefit from the implicit compression provided by exploitation of these properties, new table lookup architectures are developed. The functionality of these

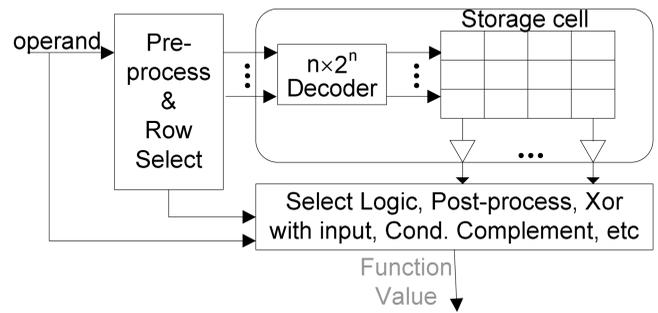


Fig. 4. Table lookup architecture.

architectures is easily described through the concept of “lookup trees.”

Lookup trees were introduced with regard to the multiplicative inverse function for odd integers modulo  $2^k$  in [5]. Properties 1, 2, and 4 described above were shown to result in substantial table size reduction, but a method and an architecture for efficient lookup were left open. The integer square function satisfies the inheritance principle, with argument normalization and appropriate conditional complementation further reducing the size of the lookup tree. In Section 2, we showed a preferred encoding allowing the discrete logarithm to satisfy and benefit from all four preceding integer function properties.

### 5.1 Table Lookup Architecture

Table lookup allows for direct conversions between binary and DLS encodings resulting in fast performance. Fig. 4 shows the generic table lookup architecture similar to that described in [15]; however, the originality of our approach is the structure of the pre- and postprocessing logic that results from the concept of the lookup tree described in [5]. In this section, we focus in detail on the example of binary-to-DLS conversion, although the methods pertain similarly to DLS-to-binary conversion. The hardware comprises three major components: the preprocess block, the postprocess block, and a ROM. The preprocess block produces the ROM address based on the input operand. After the data in the ROM is read, the postprocess block will select the correct bit fields and perform some additional processing, such as selective complementation. Two schemes for table lookup are compared here. One scheme uses a larger table supplemented by postprocess logic, while the other one uses a smaller table with both preprocess and postprocess logic.

### 5.2 Direct Lookup with Unnormalized Table Index

For the unnormalized index larger sized table implementation, we only exploit the hereditary and one-to-one mapping properties of binary-to-DLS conversion. Due to the one-to-one property, only left children of the lookup tree need to be stored. No preprocessing is required before table lookup occurs. For postprocessing, conditional complementation is required on the table output value with the input value since only left children values are stored in the table. The circuit structure and then the hardware implementation are discussed next.

The ROM structure and select logic are shown in Fig. 5. The ROM is equivalent to three-level trees. The first level forms 256 rows, where the low 8 bits ( $[a0 : a7]$ ) are used as

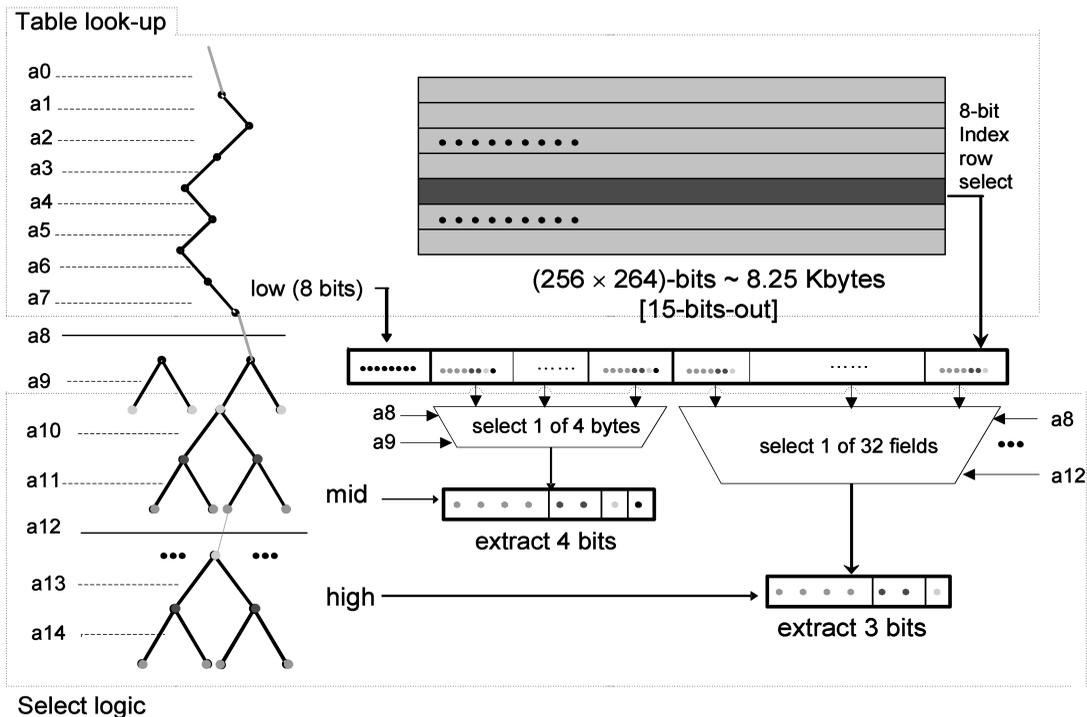


Fig. 5. Fifteen-bit table lookup architecture for  $(e, s, p)$ .

address bits. In the second level, four subtrees between levels 8 and 9 are formed as 4 bytes.  $[a8 : a9]$  are used to select one of 4 bytes. After the byte is determined,  $[a10]$  and  $[a10 : a11]$  are used to select 1 bit from the byte, respectively, while the other 2 bits are extracted directly without selection. Therefore, a total of 4 bits is extracted from the selected byte. In the third level, there are 32 subtrees between level 8 and level 12 formed as 32 7-bit fields.  $[a8 : a12]$  are used to select one of the 32 7-bit fields.  $[a13]$  and  $[a13 : a14]$  are used to select 1 bit from the selected field, respectively, while the single rightmost bit is extracted directly without selection. Therefore, a total of 3 bits is extracted from the selected 7-bit field. Finally, a 15-bit output is produced from the select logic.

The postprocessing logic for this unnormalized index table lookup scheme is simple. Since we only store left children, only 15 bits are extracted from the ROM. A one is padded to the LSB position to produce a 16-bit output. Also, 16 2-bit-input XOR gates serve as conditional complement logic, where the corresponding bit from the result of the padding and the input are connected to the inputs of the XOR gates.

### 5.3 Direct Lookup with Normalized Table Index

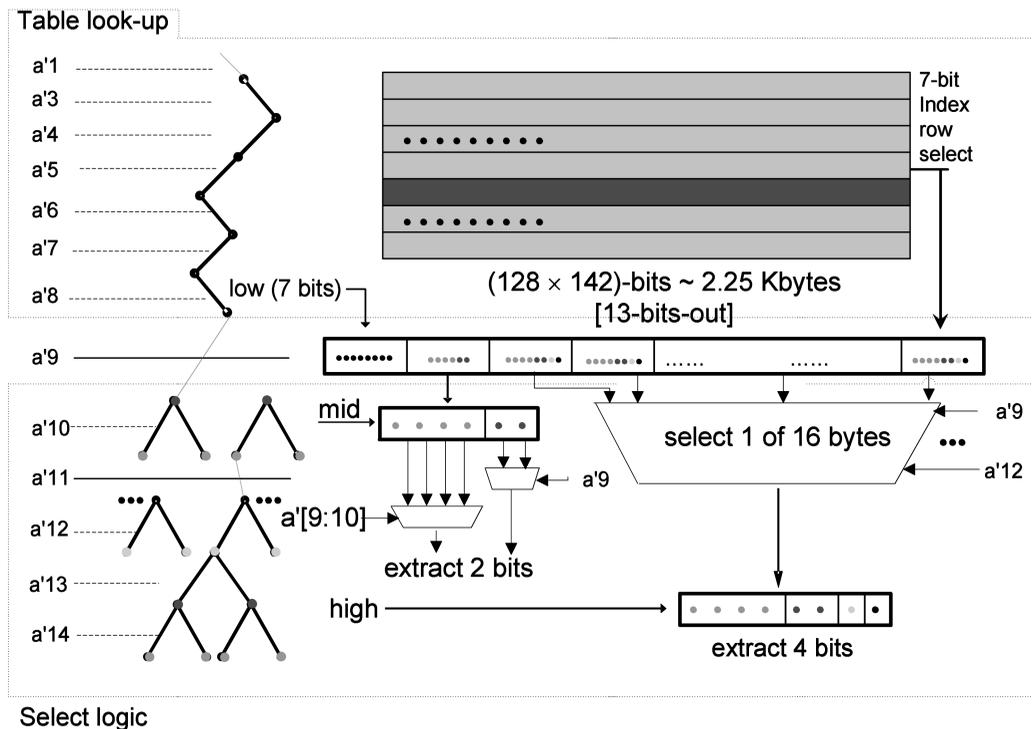
The ROM table size may be reduced by utilizing more properties of our discrete log encoding. For normalized binary-to-DLS conversion, the inheritance principle, one-to-one mapping property, normalization to odd factor argument, and conditional complementation [5] are used.

Preprocessing consists of even-power and sign bit extraction. Normalization is used to produce the  $p$  field of the DLS triple. It is accomplished by shifting to the right and counting the number of trailing zeros. In the worst case, 16 shifts are required. A divide and conquer approach is adopted in our implementation. We first shift 8 bits to the

right to check whether it is in the lower 8 bits or the higher 8 bits. Next, we shift 4 bits to the right of the selected 8-bit field from the previous step to check whether it is in the lower 4 bits or the higher 4 bits. This procedure continues until the binary exponent  $p$  of the operand is obtained. Another operation is sign extraction. The sign bit is the third bit of the normalized operand. If the sign bit is asserted, it is required to conditionally complement the normalized operand. Since normalization (odd number, no need for  $a0$ ) and sign symmetry (sign bit  $a2$  is out), the index for address and select logic in the next step are formed as  $[a'1a'3 : a'14]$  after conditional complementation.

The ROM structure and select logic are shown in Fig. 6. The ROM is equivalent to three-level trees. The first level forms 128 rows, where the low 7 bits ( $[a'1a'3 : a'8]$ ) are used as address bits. In the second level, subtrees between levels 7 and 8 are represented as a 6-bit field.  $[a'9]$  and  $[a'9 : a'10]$  are used to select 1 bit from the selected field, respectively. Therefore, a total of 2 bits are extracted from the 6-bit field. In the third level, 16 subtrees between level 7 and level 10 are formed as 16 bytes.  $[a'9 : a'12]$  are used to select one of 17 bytes.  $[a'13]$  and  $[a'13 : a'14]$  are used to select 2 bits from the selected byte, respectively, while the other 2 bits are extracted directly without selection. Therefore, a total of 4 bits is extracted from the selected 7-bit byte. Finally, a 13-bit output is formed from the select logic.

Postprocessing for the normalized index smaller table lookup scheme is more complex as compared to the larger table approach. Since normalization is performed in the preprocessing circuitry, denormalization is necessary. All bits whose index is less than the power of the original operand are padded with zeros, while all bits whose index is larger than this power are filled with lookup values. Sixteen 2-bit-input XOR gates are used for conditional complementation as described previously.

Fig. 6. Thirteen-bit table lookup architecture for  $e$ .

#### 5.4 Performance Evaluation

We described the circuits shown in Figs. 5 and 6 in a *Verilog* module using the toolset (Design Compiler and Physical Compiler) based on a standard cell library obtained from the Synopsys tutorial files [13].

Table 4 shows the comparison between the two schemes for direct lookup table conversion for  $k = 16$ . The ROM size is given in kilobytes. The core area is the area of standard cell implementation for all other logic except ROM. Both circuits have the same minimal clock period of 1.7 ns, but the larger table implementation requires one less cycle for postprocessing. Due to the extra processing before and after accessing the ROM, the normalized version of the circuit requires three clock periods of latency versus the two required for the unnormalized version; however, the ROM size is only 27 percent as large.

#### 5.5 Extension of Lookup Structure

In this section, we have investigated standard cell implementations of a novel table lookup procedure for binary-to-discrete log conversion. This method is equally applicable to realizing any integer function satisfying the inheritance principle that can then be described with a “treelike” lookup table structure.

The distinction between real and integer arithmetic in an ALU is conveniently described with reference to the

TABLE 4  
Comparison for Two Conversions

$k=16$ (wordsize)	ROM (KB)	area ( $\mu\text{m}^2$ )	Period (ns)	Latency
<i>Unnormalized</i>	8.25	21011.2	1.70	2
<i>Normalized</i>	2.25	19003.5	1.70	3

multiplication of two  $k$ -bit integer operands. The exact product fits in a  $2k$ -bit field. The real (e.g., floating point) result typically provides a normalized high-order (approximate) part with the low part rounded off. The integer result is the  $k$ -bit low-order part providing an exact result in a modular system with the modulus determined by the word size implicitly truncating the high-order part.

Integer functions that are determined modulo  $2^k$  for  $k$ -bit word results generally have properties allowing much smaller tables for exhaustive storage of  $k$ -bits-in,  $k$ -bits-out function evaluation than corresponding real valued  $k$ -bit functions. The four properties of integer functions identified in this section used in combination can fundamentally redefine and reduce the size of lookup tables for exhaustive storage. For 16-bit arguments, the advantages for integer function lookup can be as large as 32-to-1 or even 64-to-1, allowing 5 or 6 more index bits for comparable table size.

Our investigation indicates that this table lookup procedure is practical and allows for significant reductions in table size.

## 6 CONCLUSION

We have presented an alternative representation for the  $k$ -bit integers based on a discrete logarithm representation and introduced a novel DLS encoding with a one-to-one mapping between binary-encoded and DLS-encoded  $k$ -bit strings. The mapping is shown to be implementable using a new unified conversion/deconversion algorithm that employs just  $O(k)$  additions with references to a conversion lookup table having just  $k$  entries, allowing scalable implementations for  $k$  up to 128 bits or more. To illustrate the use of DLS representation, we provided a novel pipelined bit serial integer power operation for  $x^y$

employing intermediate DLS representation that returns  $x^y$  in binary using just  $O(k)$  additions without multiplications. We have also described a table lookup structure and architecture for binary/DLS conversion and presented an implementation of this structure with performance results. Favorable area and power results are obtained for the powering operation  $x^y$  and the new table lookup architecture.

## ACKNOWLEDGMENTS

This work was supported in part by the Semiconductor Research Cooperation under Contract 1399-001.

## REFERENCES

- [1] N.F. Benschop, *Multiplier for the Multiplication of at Least Two Figures in an Original Format*, US Patent 5,923,888, July 1999.
- [2] A. Fit-Florea and D.W. Matula, "A Digit-Serial Algorithm for the Discrete Logarithm Modulo  $2^k$ ," *Proc. 15th IEEE Int'l Conf. Application-Specific Systems, Architectures, and Processors (ASAP '04)*, pp. 236-246, 2004.
- [3] A. Fit-Florea, D.W. Matula, and M.A. Thornton, "Additive Bit-Serial Algorithm for the Discrete Logarithm Modulo  $2^k$ ," *IEE Electronics Letters*, vol. 41, no. 2, pp. 57-59, Jan. 2005.
- [4] A. Fit-Florea, D.W. Matula, and M.A. Thornton, "Addition-Based Exponentiation Modulo  $2^k$ ," *IEE Electronics Letters*, vol. 41, no. 2, pp. 56-57, Jan. 2005.
- [5] D.W. Matula, A. Fit-Florea, and M.A. Thornton, "Lookup Table Structures for Multiplicative Inverses Modulo  $2^k$ ," *Proc. 17th IEEE Symp. Computer Arithmetic (ARITH-17 '05)*, pp. 130-135, 2005.
- [6] L. Li, A. Fit-Florea, M.A. Thornton, and D.W. Matula, "Hardware Implementation of an Additive Bit-Serial Algorithm for the Discrete Logarithm Modulo  $2^k$ ," *Proc. IEEE CS Ann. Symp. VLSI (ISVLSI '05)*, pp. 130-135, 2005.
- [7] L. Li, M.A. Thornton, and D.W. Matula, "A Fast Algorithm for the Integer Powering Operation," *Proc. 16th ACM Great Lakes Symp. VLSI (GLSVLSI '06)*, pp. 302-307, 2006.
- [8] L. Li, A. Fit-Florea, M. Thornton, and D.W. Matula, "Performance Evaluation of a Novel Direct Table Lookup Method and Architecture with Application to 16-Bit Integer Functions," *Proc. IEEE Int'l Conf. Application-Specific Systems, Architectures, and Processors (ASAP '06)*, Sept. 2006.
- [9] N.S. Szabo and R.I. Tanaka, *Residue Arithmetic and Its Applications to Computer Technology*. McGraw-Hill Book, 1967.
- [10] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, second ed. The MIT Press, pp. 879-880, 2001.
- [11] B. Parhami, *Computer Arithmetic Algorithms and Hardware Designs*, pp. 383-384. Oxford Univ. Press, 2000.
- [12] D. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, vol. 2, second ed., pp. 441.1-466.1. Addison Wesley, 1981.
- [13] *Synopsys Design/Physical Compiler Student Guide*, 2003.
- [14] J.E. Stine, J. Grad, I. Castellanos, J. Blank, V. Dave, M. Prakash, N. Illiev, and N. Jachimiec, "A Framework for High-Level Synthesis of System-on-Chip Designs," *Proc. IEEE Int'l Conf. Microelectronic Systems Education (MSE '05)*, pp. 67-68, June 2005.
- [15] B. Parhami, *Computer Arithmetic Algorithms and Hardware Designs*, chapter 24, Oxford Univ. Press, 2000.
- [16] A. Fit-Florea, "Extending Hardware Support for Arithmetic Modulo  $2^k$ ," PhD dissertation, Dept. of Computer Science and Eng., Southern Methodist Univ., 2006.
- [17] L. Li, "Integrated Techniques for the Formal Verification and Validation of Digital Systems," PhD dissertation, Dept. of Computer Science and Eng., Southern Methodist Univ., 2006.



**Alexandru Fit-Florea** received the BS and MS degrees in computer science from "Babes-Bolyai" University, Cluj-Napoca, Romania, in 1997 and 1998, respectively, and the PhD degree in computer science from Southern Methodist University in 2005. Since 2004, he has been with Advanced Micro Devices (AMD), Sunnyvale, California, where he is currently a member of the technical staff and is working on the architecture of a next generation of microprocessors. His interests include computer arithmetic, instruction set extensions and implementation, and floating-point algorithm design.



**Lun Li** received the BS degree from Beijing Jiaotong University, Beijing, in 1997, the MS degree in electrical engineering from University of Tulsa in 2002, and the PhD degree in computer engineering from Southern Methodist University in 2006. Since 2007, he has been with Texas Instruments, where he is working on formal verification. His interests include computer arithmetic, CAD, and formal verification.



**Mitchell A. Thornton** received the BSEE degree from Oklahoma State University in 1985, the MSEE degree from the University of Texas, Arlington, in 1990, and the MSCS degree and the PhD degree in computer engineering from Southern Methodist University, Dallas, in 1993 and 1995, respectively. His industrial experience includes full-time employment at E-Systems (now L-3 Communications), Greenville, Texas, and Cyrix, Richardson, Texas, where he served in a variety of engineering positions between 1985 and 1992. From 1995 to 1999, he was a faculty member in the Department of Computer Systems Engineering, University of Arkansas and from 1999 to 2002 in the Department of Electrical and Computer Engineering, Mississippi State University. He is currently a professor of computer science and engineering and, by courtesy, electrical engineering in the Department of Computer Science and Engineering, Southern Methodist University. His research interests are in the general area of digital circuits and systems design with specific emphasis in EDA/CAD methods including asynchronous circuit and computer arithmetic circuit synthesis, formal verification/validation and simulation of digital systems, multiple-valued logic, and spectral techniques. He has served on the program committees of several conferences and symposia including general chair of the IEEE International Symposium on Multiple-Valued Logic in 2008 and cogeneral chair of the 2007 Dallas Circuits and Systems Workshop. He is a registered professional engineer in the states of Texas, Arkansas, and Mississippi and serves as a subject matter expert for the National Council of Examiners for Engineers and Surveyors (NCEES). He is also a member of the IEEE CS TAB Committee on Multiple-Valued Logic and the IEEE-USA Committee on Professional Licensure and Registration. He is a senior member of the IEEE.



**David W. Matula** received the PhD degree in engineering from the University of California, Berkeley, in 1966. He is currently a professor in the Department of Computer Science and Engineering, Southern Methodist University, Dallas. He is the author of more than 100 papers on computer arithmetic and graph algorithms. He was a codesigner of the Cyrix floating point coprocessors, and of the one Watt Geod processor employed in the OLPC. He holds 18 patents on

computer arithmetic and cellular communication systems. He was the coeditor of two special issues of the *IEEE Transactions on Computers* on computer arithmetic appearing in 1977 and 1992. He was the keynote speaker at the 16th IEEE Symposium on Computer Arithmetic in 2003. He is a member of the IEEE.

► For more information on this or any other computing topic, please visit our Digital Library at [www.computer.org/publications/dlib](http://www.computer.org/publications/dlib).