Performance Evaluation of a Novel Direct Table Lookup Method and Architecture With Application to 16-bit Integer Functions

L. Li, Alex Fit-Florea, M. A. Thornton, D. W. Matula
Southern Methodist University, {**lli,alex,mitch,matula**}**@engr.smu.edu**

Abstract

We describe several integer function properties which in combination allow direct lookup tables to be reduced in size and structure to simpler lookup trees. Our principal result is a novel table lookup method based on a mapping of a lookup tree to a row-by-column ROM with pre and post processing logic substantially reducing the table size. Our lookup architecture allows common 16-bit integer functions such as multiplicative inverse, square, and the discrete log to be realized with table size of order 2-8 Kbytes, in comparison to the 128 Kbyte size of an arbitrary 16-bits-in 16-bits-out function table. The lookup methodology is illustrated with specific development of the 16-bit integer discrete log function. Implementation for both unnormalized and normalized indices are synthesized into standard cell netlists and performance and area results are given that demonstrate their effectiveness.

1. Introduction

This paper investigates new table lookup architectures to extend the range of options for table assisted computation in optimizing an ALU design. The focus is on integer function evaluation where recent results have identified needs for new lookup architectures to exploit the potential savings available.

The distinction between real and integer arithmetic in an ALU is conveniently described with reference to the 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 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 following four properties of integer functions have been recently identified in combination to 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.

(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 2×2^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 post-processing logic to reduce the table size by another half.

(3) Normalization (separating odd and even factors): Employing a right-normalized binary integer representation, $n = i\times 2^p$ (where i 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 post-processing 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 this implicit compression, new table lookup procedures responsive to alternative "lookup tree" table architectures must be developed.

Lookup trees were introduced with regards to the multiplicative inverse function for odd integers modulo 2^k in [6]. Preceding properties (1), (2), and (4) were shown to result in substantial table size reduction, but a method and architecture for efficient lookup was 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 summarize needed background on the integer discrete-log binary representation and a preferred encoding allowing the discrete logarithm to satisfy and benefit from all four preceding integer function properties.

Our focus shifts in section 3 to the main issue of presenting an efficient architecture for implementing lookup trees for integer functions. Employing the discrete log function for illustration, our principal result is given by showing how a rectangular row-by-column bit array similar to a ROM can be designed to store a lookup tree, with the details realized in the novel selection architecture for extracting and concatenating the bits of the result into the output register. Un-normalized and normalized argument versions employing various amounts of pre and post processing logic to effect table size reduction are described. Section 4 describes comparative results of standard cell implementations of the two versions and information about the cell library employed, and Section 5 provides a brief conclusion.

2. Normalization and Discrete Log Encoding

Any positive integer has a unique factorization into odd and even-power factors, $n = i \times 2^p$, which provides a right-normalized format for binary integer representation. For integers in the "k-bit" range $[1,2^k-1]$, note that the 2^{k-2} members of the sequence $3^0,3^1,3^2,...,3^{2^{k-2}-1}$ reduce modulo 2^k to a sequence of distinct odd numbers covering half the odd numbers in $[1,2^k-1]$. The complementary values $\left|(-1)^s 3^e\right|_{2^k}$ for $0 \le e \le 2^{k-2} - 1$ cover the other half of the odd numbers where $\left|\bullet\right|_{2^k}$ denotes the standard residue modulo 2^k. 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. Benschop combined these observations in [2] noting that any integer n satisfying $1 \le n \le 2^{k-1}$ has a "modular factorization" $n = \left|(-1)^s 2^p 3^e\right|_{2^k}$. Benschop then represents every (non-zero) k-bit integer by the "discrete log" exponent triple (s,p,e). The triple is unique with $0 \le p \le k$-1 employing a minimum e always in the range $0 \le e \le 2^{k-2}-1$.

Benschop left open two fundamental questions regarding implementation of a discrete-log number system (DLS).

(1) Conversion: How do we implement the binary-integer-to-(s,p,e) triple conversion? Specifically how do we efficiently implement determination of the (s,e) pair for an odd integer i such that $i = \left|(-1)^s 3^e\right|_{2^k}$ in a scalable manner?

(2) Encoding: How do we encode the triple (s,p,e) into a word with an appropriate integer range and convenient scalability for variable word sizes?

Efficient scalable iterative solutions of the integer-to-discrete-log *conversion and deconversion questions* were presented at the algorithmic level in [1][3] with hardware implementation in [4][7]. In this paper we present two direct table lookup conversion solutions applicable for precisions up to 16 bits. Our solutions include resolution of the encoding question in a manner facilitating reducing the size of the table lookup structure with a novel lookup architecture.

The encoding employs variable length fields for the encodings of p and e and provides a one-to-one "hereditary" mapping between k-bit discrete-log numbers and k-bit unsigned binary integers. Details and proofs are beyond the scope of this paper. We utilize example tables and figures to illustrate the encoding and some of its significant properties.

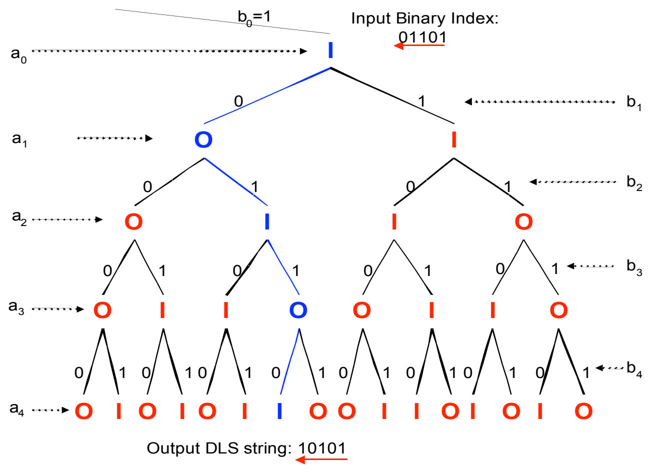
The one-to-one mapping between 5-bit discrete-log numbers comprising a 5-bit DLS and 5-bit integers is given in Table 1. The DLS bit string is partitioned as follows to determine the three exponents p, e, and s.

Consider the line in the table for DLS string 10110. The parsing begins from the right hand side determining the variable length field identifying $2^p = 2^1$ by counting zeros until the first unit bit is encountered. The next bit is a separation bit providing the logical value $s \oplus e_0$. The remaining leading bits are the 3-p bits of the exponent $0 \le e \le 2^{3-p}-1$ sufficient to determine the odd factor $i = \left|(-1)^s 3^e\right|_{2^k}$. Thus, $n = i \times 2^p$ is the integer represented with $0 \le x \le 2^5-1$ uniquely determined. In this example, $e=10_2=2_{10}$, and then s=1 is determined from e_0=0 and $s \oplus e_0 = 1$. Then $\left|(-1)^1 2^1 3^2\right|_{32} = \left|-18\right|_{32} = 14$, or $b_4 b_3 b_2 b_1 b_0$ = 01110.

The conversions for odd integers in Table 1 can be visualized by the lookup trees illustrated in Figures 1A and 1B. Navigation in Figure 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

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



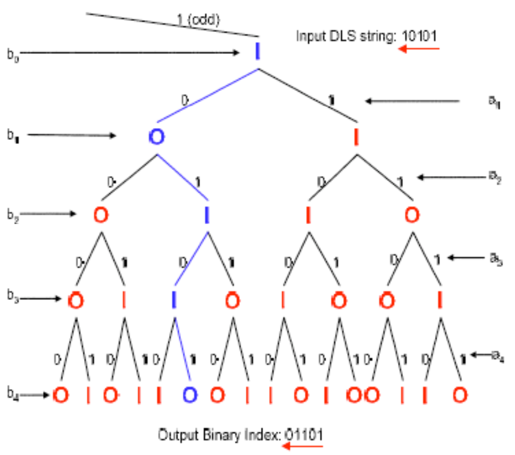


Figure 1A and 1B: Lookup Trees for Odd Integer Binary to Discrete Log Number System (DLS) Conversion and DLS to Odd Integer Binary Conversion

on the downward path. For the integer 13, employing the binary value $b_4b_3b_2b_1b_0 = 01101$ as input in Figure 1A yields the DLS string output $a_4a_3a_2a_1a_0 = 10101$. Reversing the conversion by inputting the DLS string 10101 into Figure 1B for DLS-to-binary conversion with the same navigation yields the standard binary integer output $b_4b_3b_2b_1b_0 = 01101$.

The one-to-one and conditional complementation properties hold for these conversions and are evident as symmetries in the lookup trees. With elementary pre- and post-processing logic, these properties can be employed to reduce lookup tree size for 16-bit DLS-to-binary and binary-to-DLS conversions to 2 Kbytes each.

3. Hardware Implementation

Table lookup allows for direct conversions between binary and DLS encodings resulting in fast performance. Figure 2 shows the generic table lookup architecture. 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 modular multiplicative inverse, and the square function. The hardware comprises three major components: the pre-process block, the post-process block and a ROM. The pre-process block produces the ROM address based on the input operand. After the data in the ROM is read, the post-process 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 post-process logic, while the other one uses a smaller table with both pre-process and post-process logic.

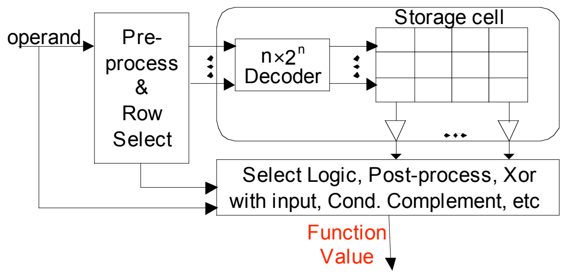


Figure 2. Table lookup architecture

3.1 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 be stored. No pre-processing is required before table lookup occurs. For post-processing, 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 Figure 3. The ROM is equivalent to 3-level trees. The first level forms 256 rows where the low 8-bits ([$a0$:$a7$]) are used as address bits. In the second level, four sub-trees between levels 8 and 9 are formed as four bytes. [$a8$:$a9$] are used to select one of four bytes. After the byte is determined, [$a10$] and [$a10$:$a11$] are used to select one bit from the byte respectively, while the other two bits are extracted directly without selection. Therefore, a total of 4 bits are extracted from the selected byte. In the third level, there are 32 sub-trees 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 one bit from the selected field respectively, while the single rightmost bit is extracted directly without selection. Therefore, a total of 3 bits are extracted from the selected 7-bit field. Finally, a 15-bit output is produced from the select logic.

The post-processing logic for this un-normalized 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 Least Significant Bit (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.

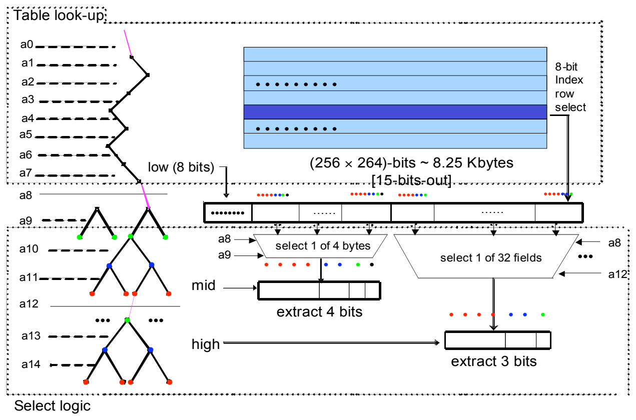


Figure 3. 15-bit table lookup architecture for (e,s,p)

3.2 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 [6] are used.

Pre-processing 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 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 right 8 bits to check whether in the lower 8 bits or the higher 8 bits. Next, we shift right 4 bits of the selected 8-bit field from the previous step to check whether 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 Figure 4. The ROM is equivalent to 3-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, sub-trees between level 7 and 8 are represented as a 6-bit field. $[a'9]$ and $[a'9:a'10]$ are used to select one bit from the selected field respectively. Therefore, a total of 2 bits are extracted from the 6-bit field. In the third level, 16 sub-trees 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 two bits from the selected byte respectively, while the other two bits are extracted directly without selection. Therefore, a total of 4 bits are extracted from the selected 7-bits byte. Finally, a 13-bit output is formed from the select logic.

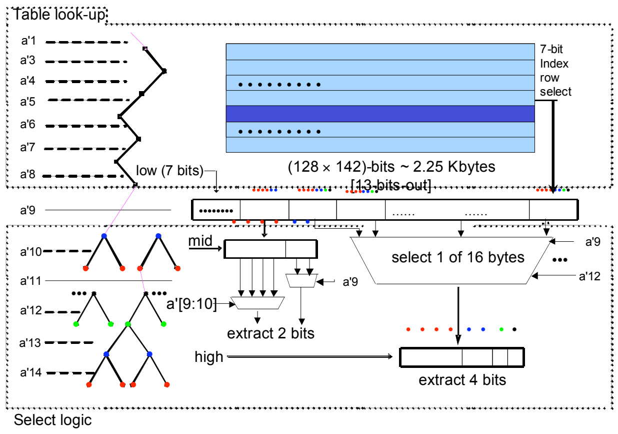


Figure 4. 13-bit table lookup architecture for *e*

Post-processing for the normalized index smaller table lookup scheme is more complex as compared to the larger table approach. Since normalization is performed in the pre-processing circuitry, de-normalization 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. 16 2-bit-input XOR gates are used for conditional complementation as described previously.

4. Experimental Results

We described the circuits shown in Figures 3 and 4 in a *Verilog* module using the tool set (Design Compiler and Physical Compiler) based on a standard cell library obtained from the Synopsys tutorial files [5].

Table 2 shows the comparison between the two schemes for direct lookup table conversion for k=16. The ROM size is given in KB. 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 1.7ns but the larger table implementation requires one less cycle for post-processing. Due to the extra processing before and after accessing the ROM, the normalized version of the circuit requires 3 clock periods of latency versus the 2 required for the unnormalized version; however, the ROM size is only 27% as large.

Table 2. Comparison for two conversions

k=16 (wordsize)	ROM (KB)	area (μm^2)	Period (ns)	Lat.
Unnormalized	8.25	21011.2	1.70	2
Normalized	2.25	19003.5	1.70	3

5. Conclusion

In this paper we have investigated standard cell implementations of a new table lookup procedure for binary-to-discrete log conversion. This method is equally applicable to realizing any integer function satisfying the inheritance principal that can be described with a "tree-like" lookup table structure. Our investigation indicates that this table lookup procedure is practical and allows for significant reductions in table size.

Acknowledgement

We thank the Synopsys Corporation for the donation of their tools and the use of their standard cell library.

6. References

[1] A. Fit-Florea, D.W. Matula, M.A. Thornton, "Additive Bit-serial Algorithm for the Discrete Logarithm Modulo 2^k," *IEE Electronics Letters,* Jan. 2005, Vol.41, No.2, pp.57-59.
[2] N.F. Benschop, "Multiplier for the multiplication of at least two figures in an original format," *US Patent* 5,923,888, July 1999.
[3] A. Fit-Florea, D.W. Matula, M.A. Thornton, "Addition-Based Exponentiation Modulo 2^k," *IEE Electronics Letters*, Jan. 2005, Vol.41, No.2, pp.56-57.
[4] L. Li, A. Fit-Florea, M.A. Thornton, D.W. Matula, "Hardware Implementation of an Additive Bit-Serial Algorithm for the Discrete Logarithm Modulo 2^k," *Proc. ISVLSI*, 2005, pp.130-135.
[5] Synopsys Design/physical Compiler Student Guide. 2003.
[6] D.W. Matula, A. Fit-Florea, M.A. Thornton, "Lookup Table Structures for Multiplicative Inverses Modulo 2^k," *Proc. 17th IEEE Symp. Comp. Arith.*, 2005, pp.130-135.
[7] L Li, M.A. Thornton, D.W. Matula "A Fast Algorithm for the Integer Powering Operation," *Proc. GLSVLSI*, 2006, pp. 302-307.