Table Lookup Structures for Multiplicative Inverses Modulo 2^k

David W. Matula* Alex Fit-Florea Mitchell Aaron Thornton†

Southern Methodist University
Dallas, Texas
matula,alex,mitch@engr.smu.edu

Abstract

We introduce an inheritance property and related table lookup structures applicable to simplified evaluation of the modular operations "multiplicative inverse", "discrete log", and "exponential residue" in the particular modulus 2^k. Regarding applications, we describe an integer representation system of Benschop for transforming integer multiplications into additions which benefits from our table lookup function evaluation procedures.

We focus herein on the multiplicative inverse modulo 2^k to exhibit simplifications in hardware implementations realized from the inheritance property. A table lookup structure given by a bit string that can be interpreted with reference to a binary tree is described and analyzed. Using observed symmetries, the lookup structure size is reduced allowing a novel direct lookup process for multiplicative inverses for all 16-bit odd integers to be obtained from a table of size less than two KBytes. The 16-bit multiplicative inverse operation is also applicable for providing a seed inverse for obtaining 32/64-bit multiplicative inverses by one/two iterations of a known quadratic refinement algorithm.

1 Introduction and Summary

Hardware integer arithmetic is generally provided for addition and multiplication modulo 2^k for k=16, 32, and possibly 64. Benschop [1] has shown a transformed binary representation that allows multiplication to be performed as an addition of "discrete logarithms". Specifically, Benschop employs the fact that every integer j in the range $[0, 2^k - 1]$ can be represented by the exponent triple (s, e, p) such that $(-1)^s 3^e 2^p \equiv j \pmod{2^k}$. Multiplication is then reduced to componentwise modular addition of the terms in Benschop's exponent triple analogous to the use of traditional logarithms for performing real-valued multiplication as a sum of the argument's logarithms.

In this paper we employ modular function notation [12] using $|n|_{2^k} = j$ to denote the congruence $n \equiv j \pmod{2^k}$ for $k \geq 1$, with the further condition that j is the standard residue for modulus 2^k satisfying $0 \leq j \leq 2^k - 1$. Thus, the exponent triple (s, e, p) for j is specified by $|(-1)^s 3^e 2^p|_{2^k} = j$.

Note that Benschop's representation is essentially a "discrete log triple *transform*". Conversion between standard binary and Benschop's exponent triples requires efficient algorithms for the exponential residue operation $|3^e|_{2^k}$ and the discrete logarithm $dlg(j)$, which is the exponential residue inverse operation (when it exists) satisfying $j = |3^{dlg(j)}|_{2^k}$. The modular multiplicative inverse $|n^{-1}|_{2^k}$ is defined for every odd integer $1 \leq n \leq 2^k - 1$ by the relation $|nn^{-1}|_{2^k} = 1$. Collectively the three unary operations of discrete log, exponential residue, and multiplicative inverse, provide a set of arithmetic operations with regard to the particular modulus 2^k that has the potential both to simplify and significantly extend standard integer arithmetic hardware. The three operations share significant fundamental properties that simplify their evaluation, in practice allowing 16-bit evaluations by relatively small new lookup table structures (e.g. less than 2 KBytes each).

Our focus in this paper is on the multiplicative inverse modulo 2^k. The discrete log and an improved exponential residue algorithm are covered in [4, 5, 8, 9]. The multiplicative inverse is particularly efficiently evaluated by the quadratic refinement formula (e.g. see [7])

$$|i^{-1}|_{2^{2k}} = ||i^{-1}|_{2^k}(2 - i|i^{-1}|_{2^k})|_{2^{2k}} \qquad (1)$$

given that we may start with a substantially sized multiplicative inverse seed (e.g. 16-bit modular inverses).

Note that Equation 1 doubles the number of bits in the modular multiplicative inverse with each iteration in a manner strikingly similar to determining a more accurate ap-

*This work was supported in part by the Semiconductor Research Corporation (SRC) grant RID-1289

†This work was supported in part by the Texas Advanced Technology Program (ATP) grant 003613-0029-2003

proximate divisor reciprocal. Recall that the Newton Raphson reciprocal refinement $\rho' = \rho(2 - y\rho)$ realizes twice the "number-of-bits-of-accuracy" where ρ is an approximation of $\frac{1}{y}$ accurate to a specified "number-of-bits". Operationally, at the bit level, the Newton-Raphson reciprocal refinement procedure employed for some floating point division implementations is an approximation process generating accurate bits of the reciprocal from left-to-right, where excess low order bits are rounded off. The important distinction herein is that the modular Equation 1 is an exact process generating the modular multiplicative inverse bits right-to-left with excess overflow bits simply truncated off the top in each iteration.

Hardware integer arithmetic units already provide addition and multiplication modulo 2^k for values of k typically including $k = 8, 16, 32$ and possibly $k = 64$. Thus, a seed lookup table for inverses modulo 2^{16} would immediately expand modular integer arithmetic capability for typical small ($k = 8$) and half word ($k = 16$) size integers. From a half word ($k = 16$) modular inverse, only one iteration of quadratic refinement employing Equation 1 would be needed to obtain a 32-bit integer modular inverse. Just two iterations would yield a 64-bit integer modular inverse. In addition to assisting in algorithms for realizing Benschop's novel integer transform representation, the inverse operation modulo 2^k can also be employed for more specific applications such as obtaining extremal rounding test cases for floating point division [10].

In Section 2 we introduce the fundamental inheritance property that simplifies the representation of all three operations: multiplicative inverse, discrete log, and exponential residue. We provide further details on Benschop's exponent triple representation to justify our focus on operations in the particular binary modulus family 2^k.

In Section 3 we show how the table lookup structure for inverses modulo 2^k can be given by a bit string of size $2^k - 1$ which can be interpreted with reference to a binary "lookup tree". We also provide *Binary Decision Diagrams* (BDDs) and flowgraph visualizations of the multiplicative inverse operation to show how the inheritance property is reflected in simpler structures in both cases. The lookup tree structure is noted to be efficient to realize and represent by an array, with access specified by the well known heap data structure indexing procedure. Symmetries in the tree reducing needed array storage size are investigated in Section 4, with a resultant table size of less than 2 KBytes sufficient for a 16-bit multiplicative inverse table.

In Section 5 we discuss circuit implementation issues. Section 6 provides a conclusion and identifies further directions for research and implementation optimizations.

2 The Inheritance Property and Operations Modulo 2^k

There are three unary operations which have the potential to simplify and extend the applications of integer arithmetic utilizing k-bit strings for typical values k=16, 32, 64, and 128. All three operations inherently employ reductions for residues modulo 2^k and share fundamental properties in their computation. These operations are the unary operations of determining the inverse $|i^{-1}|_{2^k}$ for an odd integer $1 \leq i \leq 2^k - 1$, which satisfies $|ii^{-1}|_{2^k} = 1$, the exponential residue function $|3^i|_{2^k}$ here utilizing the base 3, and the discrete logarithm $dlg(j)$, which is the appropriately defined inverse (when it exists) to the exponential residue function yielding $j = |3^{dlg(j)}|_{2^k}$.

There is an application of the exponential function $|3^e|_{2^k}$ that has motivated our interest in all three of these operations. It is readily shown that the set of odd k-bit integers is given by the set of residues modulo 2^k determined by $\{|(-1)^s 3^e|_{2^k} | s \in \{0, 1\}, 0 \leq e \leq 2^{k-2} - 1\}$. For example, for $k = 4$, note that $\{|3^e|_{16} | 0 \leq e \leq 3\} = \{1, 3, 9, 11\}$, and $\{|-3^e|_{16} | 0 \leq e \leq 3\} = \{5, 7, 13, 15\}$. Benschop [1] developed an innovative application of this fact in fashioning a representation system where each k-bit integer in $[0, 2^k - 1]$ is encoded as a triple (s, e, p) of exponents employing the modular factorization $|(-1)^s 3^e 2^p|_{2^k}$.

Conversion of standard k-bit positive integers to Benschop's exponent triples allows integer multiplication to be performed by additions and the operation of raising an integer to any power from 2 to 10 to be performed by shifts and at most a single addition/subtraction. To utilize Benschop's representation in practice it is essential to obtain efficient hardware implementable algorithms for the discrete log and exponential residue operations with regard to the particular modulus 2^k. In implementing and understanding these two operations it is helpful to also have an efficient algorithm for the multiplicative inverse modulo 2^k. All three of these unary operations satisfy an important "inheritance" property simplifying their computation.

Formalization of the inheritance property is a main contribution of this paper that is introduced in a general form.

Definition 1. *Let $f(a_{k-1}a_{k-2}\cdots a_0) = b_{k-1}b_{k-2}\cdots b_0$ be a one-to-one mapping of k-bit strings to k-bit strings defined for all $k \geq 1$. Then f satisfies the* inheritance *property and provides a k-bit hereditary function if $f(a_{k-1}a_{k-2}\cdots a_0) = b_{k-1}b_{k-2}\cdots b_0$ implies $f(a_{n-1}a_{n-2}\cdots a_0) = b_{n-1}b_{n-2}\cdots b_0$ for all $n \leq k$. That is, the n^{th} output bit b_{n-1} of $f(a_{k-1}a_{k-2}\cdots a_0)$ depends only on the low order n-bit input $a_{n-1}a_{n-2}\cdots a_0$ independent of the value of the input bits $a_{k-1}a_{k-2}\cdots a_n$.*

From the definition it follows that an arbitrary k-bit hereditary function has 2^k choices for the leading bit

$b_{k-1}, 2^{k-1}$ choices for the next bit b_{k-2}, and so on. In total, an array of $\sum_{n=0}^{k} 2^n = 2^{k+1} - 1$ bits is sufficient to fully express an arbitrary k-bit hereditary function.

The three modular operations of multiplicative inverse, discrete log, and exponential residue modulo 2^k all yield one-to-one bit-string mappings that satisfy the inheritance property. As a one-to-one mapping, the multiplicative inverse is its own inverse and the discrete log and exponential residue are inverses to each other. The discrete log and exponential residue operations are considered in [4, 5, 8, 9]. In this paper we focus on the multiplicative inverse operation modulo 2^k.

Lemma 1. *[Inheritance Lemma]*

Let $n = a_{k-2}a_{k-3}\ldots a_1\ 1$ have the multiplicative inverse $m = |n^{-1}|_{2^{k-1}} = b_{k-2}b_{k-3}\ldots b_1\ 1$. Then $i = a_{k-1}2^{k-1} + n$ has the multiplicative inverse $j = |i^{-1}|_{2^k} = b_{k-1}2^{k-1} + m$. In particular for modulus 2^k, each bit b_q (for $1 \le q \le k-1$) of the inverse $|i^{-1}|_{2^k}$ depends only on the low order bits $a_q a_{q-1}\ldots a_1$ of $i = a_{k-1}a_{k-2}\ldots a_1\ 1$.

Proof. Let $|ii^{-1}|_{2^k} = 1$. Reducing both sides modulo 2^{k-1} we obtain

$1 = |ii^{-1}|_{2^k} = ||i|_{2^{k-1}}|i^{-1}|_{2^{k-1}}|_{2^{k-1}} = |n|i^{-1}|_{2^{k-1}}|_{2^{k-1}}$.

So $|i^{-1}|_{2^{k-1}}$ is the modular inverse of n, and $|i^{-1}|_{2^{k-1}} = m = b_{k-2}b_{k-3}\ldots b_1\ 1$. Thus, appending the leading bit a_{k-1} to the string $a_{k-2}a_{k-3}\ldots a_1\ 1$ impacts only the leading bit b_{k-1} of the inverse $b_{k-1}b_{k-2}\ldots b_1\ 1$, with the trailing bits $b_{k-2}b_{k-3}\ldots b_1\ 1$ *inherited* from the inverse $|(a_{k-2}a_{k-3}\ldots a_1\ 1)^{-1}|_{2^{k-1}} = b_{k-2}b_{k-3}\ldots b_1\ 1$. □

3 Lookup Trees for Inverses Modulo 2^k

The inheritance property allows us to realize a lookup table array with reference to a binary tree structure significantly reducing the necessary table storage in comparison to a $(k \times 2^k)$-bit direct lookup table. We have noted that any k-bit hereditary function can be expressed as an array of total size $2^{k+1} - 1$ bits. For 5-bit modular inverses where the low order bit of the input word and the output word is fixed at unity, we obtain a 4-bit hereditary function that may be expressed by the $2^{4+1} - 1 = 31$ bit array $T_5 = 1\,|\,01\,|\,0101\,|\,01101001\,|\,0110010101011001$. The interpretation of this bit string is provided by recognizing the string as an array form in the fashion of heap indexing of a full binary tree, which we term the multiplicative inverse lookup tree.

Definition 2. *The binary multiplicative inverse lookup tree T_k is a $(2^k - 1)$ vertex binary tree where at depth n, for $1 \le n \le k$, the left edges are labeled by $a_n = 0$ and right edges labeled by $a_n = 1$. The table lookup index $i = a_{k-1}a_{k-2}\ldots a_1 1$ read right-to-left (low order bit first) directs a path down from the root. For $2 \le n \le k$, the vertex reached by the (n-1)-bit path $a_1 a_2 \ldots a_{n-1}$ is labeled by the leading bit b_{n-1} of the multiplicative inverse determined from $|i^{-1}|_{2^n} = b_{n-1}b_{n-2}\ldots b_1 1$.*

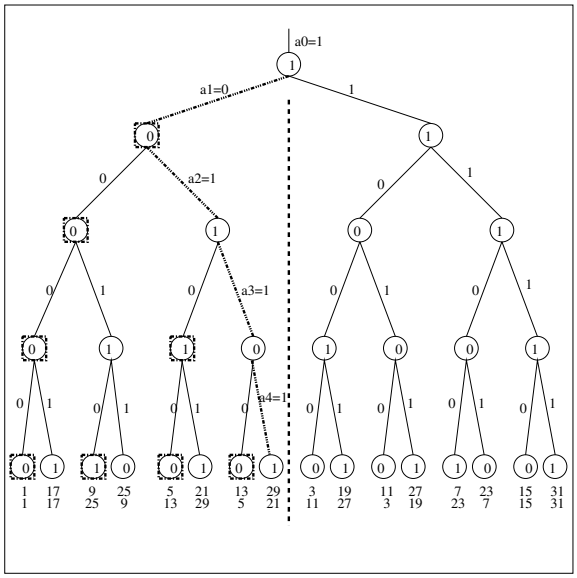

Figure 1. The binary modular multiplicative inverse lookup tree T_5. Values on the eight highlighted vertices on the left determine all vertex values by appropriate complementation.

Figure 1 shows the tree T_5 providing the inverses $|i^{-1}|_{32}$ for all odd $1 \le i \le 31$. The highlighted path for $i = a_4a_3a_2a_11 = 11101 = 29$ follows edges from the root labeled $a_1 = 0, a_2 = 1, a_3 = 1$, and $a_4 = 1$, with the successive vertex labels starting from the root labeled $b_0 = 1$ concatenated from right-to-left giving $|29^{-1}|_{32} = 10101 = 21$. Note further that following the path dictated by $a_4a_3a_2a_11 = 21$ yields the output $b_4b_3b_2b_11 = 29$ as necessary for $|(|i^{-1}|_{32})^{-1}|_{32} = i$. Figure 1 includes the values of i and $|i^{-1}|_{32}$ at all leaves to further illustrate the lookup tree structure.

The storage and retrieval array indexing method we employ is derived from the indexing structure used for the "heap" data structure. Specifically, for a complete binary tree such as T_k, the $2^k - 1$ vertices may have their single bit labels stored in a one dimensional bit array of length $2^k - 1$ bits, where the q^{th} bit of the array has its left child in location $2q$ and right child in location $2q + 1$, with parent at $\lfloor \frac{q}{2} \rfloor$. The 31 bit array for T_5 is then $1\,|\,01\,|\,0101\,|\,01101001\,|\,0110010101011001$. In this binary tree array format, the lookup tree contains the full lookup table for T_k in just $2^k - 1$ bits, with the table size just under 32 bytes for $k = 8$ bit integers and to a manageable size

of just under 8 Kilobytes for $k = 16$ bit integers. Further results on the heap data structure can be found in most data structure and algorithm texts, e.g. [3].

In array form note that the 4-bit modular inverse tree is $T_4 = 1|01|0101|01101001$. In general the inheritance property assures that the array form of T_{k-1} is simply the $(2^{k-1}-1)$-bit prefix of the (2^k-1)-bit array for T_k for any k. Thus, the multiplicative inverse function modulo 2^k for all k leads to a universal unique labeling of the full binary tree to infinite depth.

The lookup tree structure described here is implemented as a bit-string that can be stored in a standard row by column memory structure with multiplexers employed to select appropriate bit fields to form the modular inverse. The lookup tree indexing structure allows for the determination of the multiplexing operations needed to extract a specific bit field corresponding to a multiplicative modular inverse operation as will be shown in Section 4.

For our purposes, the lookup tree to a fixed depth (e.g. $k = 16$) is considered to be precomputed and stored. This is analogous to a direct reciprocal lookup table for division algorithms. Thus, we are considering a hardware implementation of the data in the tree as may be stored in a ROM or synthesized. If logic synthesis is employed to store the lookup structure, we believe the area reduction obtained by the lookup tree design is an important first step that can then be followed by logic synthesis tools.

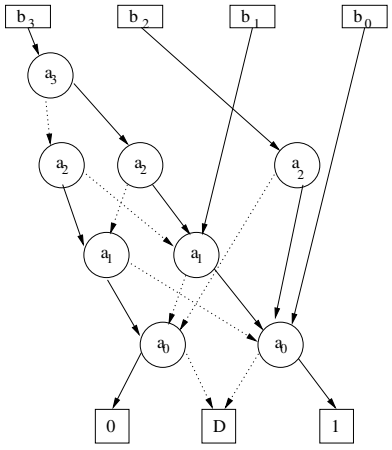

Figure 2. BDD Representing a 4-bit Multiplicative Modular Inverse Function

Figure 2 contains an alternative data structure for the 4-bit modular multiplicative inverse function known as a *Binary Decision Diagram* (BDD) [2]. The BDD represents the Boolean binary functions for each bit b_n in the multiplicative modular inverse word $b_3b_2b_1b_0$ that corresponds to $a_3a_2a_1a_0$. The edges shown as dotted lines represent paths that are traversed when the vertex variable a_n is 0-valued and the solid directed edges indicate $a_n = 1$. To determine the b_n value, a path is traversed from an initial vertex to a terminal vertex. In Figure 2, there are three terminal vertices labeled with '0', '1', or 'D' where 'D' indicates a "don't care" value that corresponds to $a_3a_2a_1a_0$ being an even value. The BDD clearly shows that each b_q value depends only on a_n values for $n \leq q$ and thus provides a method for visualizing the inheritance property. The structure we propose here for describing the modular multiplicative inverse lookup table is also a tree structure; however, the differences between the lookup tree and the BDD are apparent when comparing Figures 1 and 2.

The method described here based on lookup trees differs from the approaches in [6] and [11]. In both of these approaches, graphs or trees are used whose structure is directly related to the resulting circuit structure. In [11], BDDs are used to specify the topology of a high fan-in transistor circuit for implementing multiplier-accumulator circuits in an adiabatic style. The method described in [6] is more similar to our approach since a transistor circuit is ultimately realized in a ROM structure; however, this method also initially uses the structure of a tree as the basis for the topology of a dynamic logic block and then employs graph reduction rules for minimization of the block. The approach in [6] is more general than our approach since it may be used for any arbitrary logic function.

A third alternative to visualize the properties of the modular multiplicative inverse function is through the use of a flow graph [13]. Figure 3 contains a diagram of the flow graph for a 6-bit modular inverse table. The values at the left side of the figure represent 6-bit values i for which it is desired to compute the respective modular inverses i^{-1} such that $|ii^{-1}|_{2^k} = 1$. The values at the right side of the figure are the corresponding 6-bit modular inverses i^{-1}. This depiction of the modular inverse operation is useful for visualizing the one-to-one mapping nature of the function. The flow graph also illustrates the inheritance property by showing how modular inverse tables for k-bit values depend only on two tables of $(k-1)$-bit values. As an example, the first stage or column of the flow graph consists of four 4-bit modular multiplicative inverse operations and the second stage consists of two 5-bit inverse operations leading to the final 6-bit inverse operation.

4 Lookup Tree Storage Reduction

There are two straightforward symmetry properties in the binary tree T_k that provide storage size reduction by a factor of four.

Observation 1. *[Reflective Complementarity] For any* $k \geq 2, |(2^k - i)^{-1}|_{2^k} = 2^k - |i^{-1}|_{2^k}$.

Note that the two's complement of an odd integer

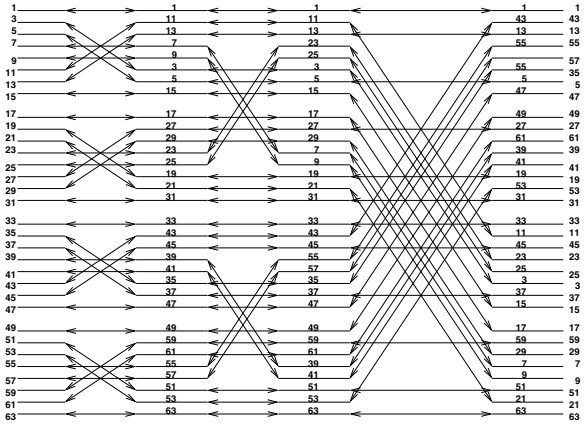

Figure 3. A flow graph for 6-bit modular inverses.

is just the one's complement of all bits except the least significant bit $a_0 = 1$. The reflective complementarity of T_5 about the middle noted in Figure 1 for T_5 holds in general for T_k as a consequence of Observation 1. Reflective complementarity allows us to compress the 31-bit array for T_5 into a 15-bit array T_5^C. In particular note that $q_1|q_2q_3|q_4q_5q_6q_7|q_8 \ldots q_{15}|q_{16} \ldots q_{31} = q_1|q_2\overline{q}_2|q_4q_5\overline{q}_5\overline{q}_4|q_8q_9q_{10}q_{11}\overline{q}_{11}\overline{q}_{10}\overline{q}_9\overline{q}_8|q_{16} \ldots q_{23}\overline{q}_{23} \ldots \overline{q}_{16}$ with $c_1|c_2c_3|c_4 \ldots c_7|c_8 \ldots c_{15} = q_2|q_4q_5|q_8 \ldots q_{11}|q_{16} \ldots q_2$. The bit sequence for T_5^C is then $0|01|0110|01100101$.

Observation 2. *[Sibling Complementarity] For any $k \geq 2$, every right child of a vertex of T_k has a one bit label that is the complement of the label of the left child of that vertex.*

Proof. This is a consequence of the fact that every odd integer i with $1 \leq i \leq 2^k - 1$ has a distinct inverse modulo 2^k. □

Sibling complementarity allows us to further compress the 15-bit array T_5^C to a 7-bit array T_5^l. This may be visualized by viewing the left subtree in Figure 1, where each of the 7 vertices (deleting the bottom row) has as its new label the label currently on its left child vertex. In particular for the array T_5^C, $c_1|c_2c_3|c_4c_5c_6c_7|c_8 \ldots c_{15} = c_1|c_2\overline{c}_2|c_4\overline{c}_4c_6\overline{c}_6|c_8\overline{c}_8c_{10}\overline{c}_{10} \ldots c_{14}\overline{c}_{14}$. So then the *left child* reduced tree T_5^l is given by the seven-bit array $l_1|l_2l_3|l_4l_5l_6l_7 = c_2|c_4c_6|c_8c_{10}c_{12}c_{14}$ and is obtained by setting $l_n = c_{2n}$ for $1 \leq n \leq 7$. The bit sequence for T_5^l is then $0|01|0100$.

Observations 1 and 2 provide elementary storage reduction techniques removing the observed redundancies in the data of the tree T_k.

Observation 3. *[Left child reduced tree] The labels for all vertices of T_k can be determined from the labels on the set of 2^{k-2} left children in the left subtree of T_k.*

The eight left children in the left subtree are highlighted by the squares around the vertices in Figure 1. It follows from Observation 2 that a left-child-compressed lookup tree array T_{16}^l for 16-bit integer inverses modulo 2^{16} consumes only 2 Kilobytes of storage.

In the following, we outline the decompression (decoding) logic needed to obtain each output bit b_{n-1} for $k \geq n \geq 3$ from the left child reduced tree for input $i = a_{k-1}a_{k-2} \ldots a_1$. First, conditionally complementing the input bit string based on a_1 to obtain an $(n-3)$-bit table index $a'_{n-2}a'_{n-3} \ldots a'_2$.

- table index $= a'_{n-2}a'_{n-3} \ldots a'_2$ equals:

$$\begin{cases} a_{n-2}a_{n-3} \ldots a_2 & \text{if } a_1 = 0, \\ \bar{a}_{n-2}\bar{a}_{n-3} \ldots \bar{a}_2 & \text{if } a_1 = 1. \end{cases}$$

The index $a'_{n-2}a'_{n-3} \ldots a'_2$ is used to read down from the root of T_k^l to determine the left-subtree left-child level $(n-1)$ bit $l(n-1)$ which is conditionally complemented by a'_{n-1} to obtain the level $(n-1)$ left-subtree bit of T_k^C denoted by $c(n-1)$.

- table output bit $= l(n-1)$.
- decoded left subtree bit $c(n-1)$ is:

$$\begin{cases} l(n-1) & \text{if } a'_{n-1} = 0, \\ \bar{l}(n-1) & \text{if } a'_{n-1} = 1. \end{cases}$$

The left subtree bit $c(n-1)$ is then conditionally complemented by a_1 to obtain the output bit b_{n-1}.

- decoded output bit $= b_{n-1}$ $\begin{cases} c(n-1) & \text{if } a_1 = 0, \\ \bar{c}(n-1) & \text{if } a_1 = 1 \end{cases}$

A combinational logic relation also exists between the grandchildren of each vertex in T_k. Figure 4 shows the generic two level subtree labeling that holds for every two-level subtree within the left subtree of T_k for all k. This allows our T_k^l to be further compressed by 33% with only the cost of another XOR gate applied to the table output.

Appropriately incorporating the relation of Figure 4 for reduction at every other level, the lookup tree for 16 bit integer multiplicative inverses modulo 2^{16} need consume only $1\frac{3}{8}$ Kbytes of storage. This further storage reduction can be useful for multiple tables for single instruction multiple data (SIMD) integer instruction sets such as the MMX operations in the x86 processors. The MMX instructions allow 4 parallel 16 bit integer addition or multiplication operations to be executed concurrently with the input and output

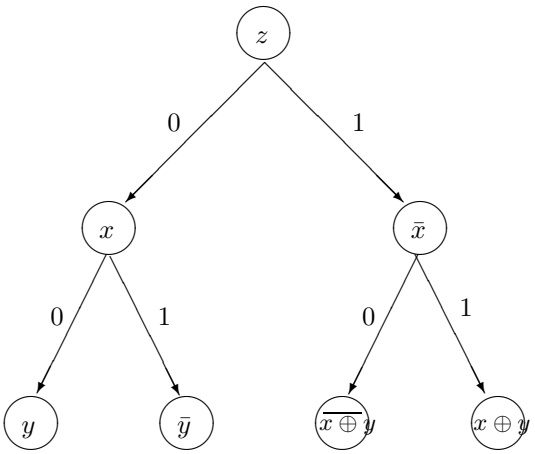

Figure 4. Generic 2-level subtree labeling in the left subtree of T_k

stored in 64 bit words partitioned into four 16-bit integer sub-words. Employing four (identical) compressed lookup trees for concurrent lookup of four 16-bit integer inverses will then employ a total of only $5\frac{1}{2}$ Kilobytes. This would allow a feasible one or two cycle SIMD implementation of the multiplicative inverse for four concurrent 16-bit integers by this direct lookup procedure.

5 Circuit Implementation Issues

In implementing the modular inverse table, a considerable amount of circuit area can be saved by utilizing symmetries and other properties that are present in the table as discussed in Section 3.

Figure 5 provides a circuit realization for inverses modulo 32 that is equivalent to the lookup tree T_5 of Figure 1. The logic circuit shown in Figure 5 is considerably simplified by the observation that every odd integer $\{1, 3, 5, 7\}$ is its own inverse modulo 8, i.e. $b_n = a_n$ for $0 \leq n \leq 2$.

The logic circuit for T_5 takes on greater significance with the observation that this circuit provides the 5 low order bits of the inverse modulo 2^k for any $k \geq 5$. Thus, only the inverse bits $b_{k-1}b_{k-2}\ldots b_5$ need be obtained from a lookup process and/or further circuitry.

The 4-bit modular inverse operation can be directly implemented as a combinational logic circuit consisting of a single 3-input XOR gate. This circuit corresponds to all but the leading bit portion of the circuit in Figure 5 that includes the bottom XOR gate and the three pass-through lines for the least significant bits. The most significant bit in the 4-bit inverse circuit is the parity of the 3 most significant bits in the argument value. The remaining three least significant bits of the inverse value are identical to the three least

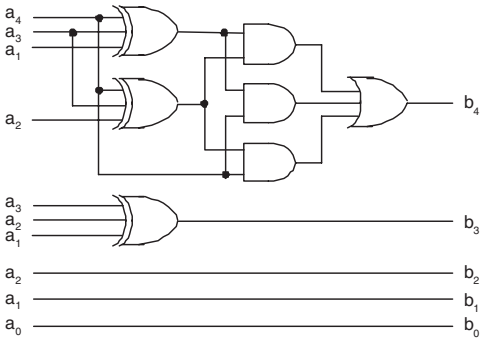

Figure 5. A combinational logic circuit for 5-bit modular inverses.

significant bits of the argument value. The behavior of this circuit is shown in the flow graph in Figure 3 by examining the top eight inputs at the left side of the flow graph and considering stage 1 only.

Likewise, the 5-bit modular inverse circuit can be implemented using 2 3-input XOR gates and the equivalent of a full-adder circuit as shown in Figure 5. Inputs to the carry-out portion of the full adder are computed as the parity of groups of 3 bits in the input words and the output is used to form the most significant bit in the inverse value.

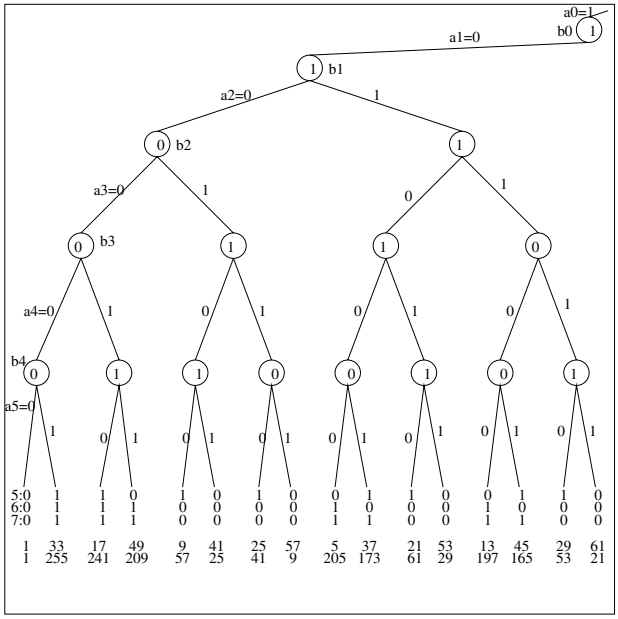

Figure 6. A reduced format for T_8.

It is possible to continue to build purely combinational logic circuits to compute the modular inverses for words of 6 bits or greater. However, the number of logic levels increases and the delay penalty may make this approach

impractical. For this reason, circuits of 6 or more bits are implemented using a lookup table for the most significant bits in the sixth position and higher. The circuit of Figure 5 is used to compute the 5 least significant bits in parallel with the lookup process for the most significant output bits $b_{k-1}b_{k-2}\cdots b_5$. We first describe this approach for the cost of an 8-bit modular inverse table. It is instructive to consider the tree T_8 of 8-bit integer modular inverses, since 8-bits is often considered a typical small or half word integer type.

Observations on Circuitry for Inverses Modulo 2^8

Figure 6 shows the left subtree of the root of T_8, where the 16 final two-level subtrees are each denoted as leaves at level five with the c_5, c_6, c_7 triple on each leaf denoting the z, x, y values to be expanded as in Figure 4. Thus for 8-bit inverses, the low order 5 bits can be obtained from the circuit of Figure 5, and a 4-bits-in 3-bits-out lookup into a 48 bit word can be used to obtain the z, x, y bits for determining the leading output bits $b_7b_6b_5$ according to Figure 4 and Observation 1, with appropriate complementation included to cover the right subtree output corresponding to $a_1 = 1$. Thus the 8-bit modular inverse circuit is composed of three types of circuitry:

1. a 5-bit modular inverse circuit as shown in Figure 5
2. a 4-bit input, 3-bit output lookup table realized as a 48 bit word.
3. a small amount of external logic used to decode the lookup table content as indicated in Figure 4.

If a lookup tree is constructed for the modular inverse for the case where n=8 bits, the 16 lowest subtrees (of the left subtrees of the root) would all have a structure identical to that shown in Figure 4. For this reason, the lookup table can be quite compact in that less than $\frac{1}{16}$ of the bits are required to be stored as compared to an exhaustive direct lookup table that would require 128 7-bit words. The resulting reduced table is the 4-bits in, 3-bits out lookup table illustrated in the leaves of the tree in Figure 6. A small amount of extra decoding logic is required if this reduced table is used. The additional logic essentially determines which of the 16 subtrees is currently being accessed and provides for decoding the values in the lookup table where appropriate.

This allows for a very compact modular inverse circuit to be realized that is a combination of a reduced lookup table with decoding circuitry and a direct circuit implementation for low order bits. In particular, the lookup table and decoder portions of the circuit are used to generate the 3 most significant bits $b_7b_6b_5$ of the inverse and a direct circuit implementation of the table (Figure 5) is used to generate the 5 least significant bits $b_4b_3b_2b_1b_0$.

Observations on Circuitry for Inverses Modulo 2^{16}

In the case of a 16-bit modular inverse circuit, the implementation technique described for the 8-bit case can be extended. A block diagram of this circuit with the table stored in a roughly square row by column format is shown in Figure 7. As is done in the 8-bit version of the circuit, the 5 least significant bits are obtained using the circuit in Figure 5 and the remaining 11 bits are computed from a compressed lookup table with decoding logic. The compressed lookup table is indexed by 7 conditionally complemented bits of the argument value, $a'_8a'_7a'_6a'_5a'_4a'_3a'_2$. The content of the lookup table is 128 lines that each represent a subtree in the 16-bit lookup tree. Each of the 128 lines in the lookup table contains the $b'_5b'_6b'_7b'_8b'_9$ bits and all possible pairs of the $b'_{10}b'_{11}$ bits (2 pairs), all possible pairs of the $b'_{12}b'_{13}$ bits (8 pairs), and all possible pairs of the $b'_{14}b'_{15}$ bits (32 pairs). This leads to a total of $32 \times 2 + 8 \times 2 + 2 \times 2 + 5 = 89$ bits of storage per line. The selection of the correct pair of bits for the cases of $b'_{10}b'_{11}, b'_{12}b'_{13}$, and $b'_{14}b'_{15}$ is accomplished through the use of multiplexers with control line inputs set to the lower order a'_i bits. After the appropriate $b'_5b'_6b'_7b'_8b'_9b'_{10}b'_{11}b'_{12}b'_{13}b'_{14}b'_{15}$ sub-word is obtained from the lookup table line, decoding logic is applied to compute the $b_5b_6b_7b_8b_9b_{10}b_{11}b_{12}b_{13}b_{14}b_{15}$ bits. The decoding logic is used to account for which side of the lookup tree that the value is stored in and to exploit the complementation symmetry described in Section 2. Only 128 subtrees are required to be stored due to the same arguments about modular inverse properties and symmetries that were described in the previous section. The total amount of required storage is approximately 11 bytes of storage for each of the 128 lines in the lookup table. Therefore, for a 16-bit modular inverse circuit, a lookup table size of approximately 1.375 KB is required. This is a significant amount of reduction as compared with the 60 KB table that would be required in a naïve direct lookup table implementation that corresponds to storing 2^{15} 15-bit words.

In terms of performance, the circuit structure as shown in Figure 7 allows for several tradeoffs. The decoding logic at the output of the circuit can be simplified by increasing the word size in the lookup table requiring at most the additional step of conditional complementation. The dotted line in Figure 7 indicates where the circuit could have pipeline latches inserted allowing for single-cycle throughput. The circuit as shown would likely require only two processor cycles to complete since the portions of the circuit to the left and right of the dotted line each depend on approximately four to six levels of logic in terms of delay. Further performance enhancement is given by the fact that the control

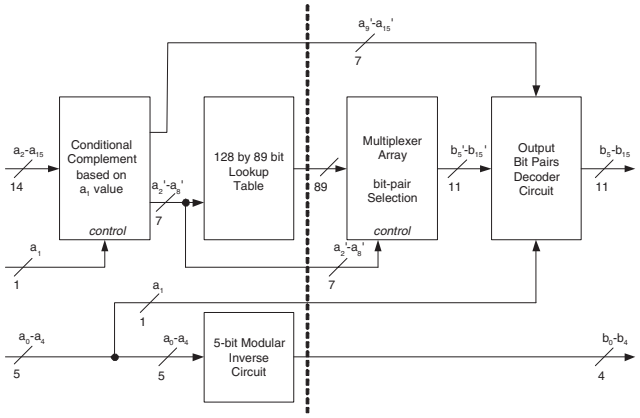

Figure 7. Block Diagram of 16-bit Modular Inverse Circuit

lines for the multiplexer array are stabilized during the time the lookup table is accessed. The critical path through the circuit is set by the $32:1$ (2-bit data inputs and output) multiplexer and XOR gate for decoding noted in Figure 4. This is the widest multiplexer in the array and it is responsible for selecting the appropriate $b'_{14}b'_{15}$ bit pair from the word retrieved from the lookup table.

6 Conclusions

The initial work we describe here is part of an ongoing investigation of the operations multiplicative inverse, discrete log, exponential residue, and squaring in the particular domain of modular arithmetic with modulus 2^k. We have introduced a new lookup tree table format applicable to lookup tables of inverses modulo 2^k. The table is realized as a binary lookup tree which is universal in the sense that the complete tree to depth k provides all inverses modulo 2^k. We also describe symmetry properties of the lookup tree that allow table size reduction supplemented by selection and decoding logic to obtain the desired lookup value.

In this paper, logic design techniques with particular emphasis on precomputed stored tables for the moduli 2^8 and 2^{16} were investigated. The opportunities to exploit symmetry within the lookup tree structure provides a classic case of space versus delay investigation. The tradeoff between the required circuitry and delay is present in the circuit of Figure 6 by changing the size of the lookup table which, in turn, impacts the corresponding amount of decoding circuitry. Further refinement and simplification of the logic circuits within the blocks of Figure 7 is ongoing.

References

[1] Benschop N. F., "Multiplier for the multiplication of at least two figures in an original format", *US Patent Nr. 5,923,888*, July 13, 1999.

[2] Bryant, R.E., "Graph-based Algorithms for Boolean Function Manipulation", *IEEE Trans. Comp.*, 35(8), 1986, pp. 677-691.

[3] Cormen, T.H., Leiserson, C.E., and Rivest, R.L., *Introduction to Algorithms*, The MIT Press and Mc-Graw-Hill, 1991, 5th edition.

[4] Fit-Florea, A., Matula, D.W., and Thornton, M.A., "Addition-based exponentiation modulo 2^k", *IEE Electronics Letters*, January, pp. 56-57, 2005.

[5] Fit-Florea, A., Matula, D.W., and Thornton, M.A., "Additive bit-serial algorithm for discrete logarithm modulo 2^k", *IEE Electronics Letters*, January, pp. 57-59, 2005.

[6] Jullien, G.A., Miller, W.C., Grondin, R., Del Pup, L., Bizzan, S.S., and Zhang, D., "Dynamic Computational Blocks for Bit-Level Systolic Arrays", *IEEE Journal of Solid-State Circuits*, 29(1), 1994, pp. 14-22.

[7] Knuth D.E., *Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Addison Wesley, 1997 3rd edition.

[8] Li, L., Fit-Florea, A., Thornton, M.A., and Matula, D.W., "Hardware Implementation of an Additive Bit-Serial Algorithm for the Discrete Logarithm Modulo-2^k", *IEEE Symposium on VLSI*, May 2005, (to appear).

[9] Li, L., Fit-Florea, A., Thornton, M.A., and Matula, D.W., "A New Binary Integer Number System with Simplified Hardware Support", *Proc. IEEE* 15^{th} *International Conference on Application-specific Systems, Architectures and Processors*, July 2005, (submitted).

[10] McFearin, L.D. and Matula, D.W., "Generation and Analysis of Hard to Round Cases for Binary Floating Point Division", *Proc.* 15^{th} *IEEE Symposium on Comuter Arithmetic*, 2001, pp. 119-127.

[11] Suvakovic, D. and Salama, A.T., "Energy Efficient Adiabatic Multiplier-Accumulator Design", *Journal of VLSI Signal Processing*, 33, 2003, pp. 83-103.

[12] Szabó, S. and Tanaka, R.I., *Residue Arithmetic and Its Applications to Computer Technology*, McGraw-Hill Book Co., 1967.

[13] Thornton, M.A., Drechsler, R., and Miller, D.M., *Spectral Techniques in VLSI CAD*, Kluwer Academic Publishers, 2001.