A SURVEY OF SIMULTANEOUS BINARY MULTIPLICATION

BY

JOSEPH LARRY VOYER

A Research Report Presented in Partial Fulfillment of the Requirements for the Degree MSE

FLORIDA TECHNOLOGICAL UNIVERSITY

August 1972
PREFACE

About three years ago, while working for General Electric Co. in Daytona Beach, I was confronted with the task of designing a 16 x 16 bit multiplier with a throughput rate of 80 nanoseconds. The only knowledge of binary multiplication, that I had at the time, was a textbook knowledge of the standard Booth and Booth algorithm. This project started my interest in the area of binary arithmetic algorithms and I have since been collecting all the articles on the subject that I could obtain. This paper represents the first time that I have been able to sit down and read any of them.

The average time spent reading, digesting, verifying, and working examples for each paper was approximately four hours. This implies that the average engineer would be hard pressed to make a decent literature search and digest the data in time for it to be of any usefulness on a tight deadline project. Since tight deadlines are the norm rather than the exception, a need exists for somebody to collate the available data and present it in an understandable form. This report fills my own personal needs in this regards. I hope that it will also be useful to my fellow engineers.
ACKNOWLEDGEMENT

I wish to thank Dr. Ling of the IBM Research Laboratory for his prompt and courteous response to my questions on his method. I wish to thank Dr. Benjamin Patz, my Report advisor, for his helpful comments and for guiding this report to its present form. I also wish to thank my wife who helped me with the bulk of the typing.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREFACE</td>
<td>iii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENT</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>NON - ITERATIVE MULTIPLY METHODS</td>
<td>3</td>
</tr>
<tr>
<td>PRIOSTE MULTIPLIER</td>
<td>5</td>
</tr>
<tr>
<td>CARRY SAVE MULTIPLIER</td>
<td>7</td>
</tr>
<tr>
<td>RICHARDS' METHOD</td>
<td>12</td>
</tr>
<tr>
<td>THE DADDAN MULTIPLIER</td>
<td>16</td>
</tr>
<tr>
<td>THE WALLACE MULTIPLIER</td>
<td>19</td>
</tr>
<tr>
<td>LING'S METHOD</td>
<td>30</td>
</tr>
<tr>
<td>CHEN'S METHOD</td>
<td>38</td>
</tr>
<tr>
<td>MITCHELL'S METHOD</td>
<td>48</td>
</tr>
<tr>
<td>CONCLUSION</td>
<td>63</td>
</tr>
<tr>
<td>LIST OF REFERENCES</td>
<td>67</td>
</tr>
<tr>
<td>BIBLIOGRAPHY I</td>
<td>68</td>
</tr>
<tr>
<td>BIBLIOGRAPHY II</td>
<td>72</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>BLOCK DIAGRAM OF THE PRIOSTE MULTIPLIER</td>
<td>6</td>
</tr>
<tr>
<td>2.</td>
<td>MULTIPLICATION OF TWO 5-BIT NUMBERS</td>
<td>8</td>
</tr>
<tr>
<td>3.</td>
<td>BLOCK DIAGRAM OF 5 X 5 MULTIPLIER USING CARRY SAVE TECHNIQUE</td>
<td>9</td>
</tr>
<tr>
<td>4.</td>
<td>MULTIPLICATION OF TWO 8-BIT NUMBERS</td>
<td>14</td>
</tr>
<tr>
<td>5.</td>
<td>BLOCK DIAGRAM OF 8 X 8 BIT MULTIPLY USING RICHARDS' METHOD</td>
<td>15</td>
</tr>
<tr>
<td>6.</td>
<td>BLOCK DIAGRAM OF 5 X 5 DADDA MULTIPLIER</td>
<td>17</td>
</tr>
<tr>
<td>7.</td>
<td>COMPARISON OF RICHARDS AND WALLACE TREE STRUCTURES</td>
<td>22</td>
</tr>
<tr>
<td>8.</td>
<td>COMPARISON OF RICHARDS AND RECODED WALLACE TREE STRUCTURES</td>
<td>23</td>
</tr>
<tr>
<td>9.</td>
<td>BLOCK DIAGRAM FOR WALLACE SCHEME (40 X 40)</td>
<td>26</td>
</tr>
<tr>
<td>10.</td>
<td>BLOCK DIAGRAM FOR WALLACE MULTIPLIER EXAMPLE</td>
<td>28</td>
</tr>
<tr>
<td>11.</td>
<td>ARITHMETIC EXAMPLE OF WALLACE MULTIPLIER</td>
<td>29</td>
</tr>
<tr>
<td>12.</td>
<td>8 X 8 BIT MULTIPLICATION PARALLELOGRAM</td>
<td>39</td>
</tr>
<tr>
<td>13.</td>
<td>BASE 2 LOGARITHM CURVE</td>
<td>51</td>
</tr>
<tr>
<td>14.</td>
<td>BASE 2 LOGARITHM CURVE (APPROXIMATION)</td>
<td>52</td>
</tr>
<tr>
<td>15.</td>
<td>EXAMPLE OF MITCHELL'S MULTIPLICATION METHOD</td>
<td>56</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ENCODING TABLE FOR THE WALLACE MULTIPLIER</td>
<td>24</td>
</tr>
<tr>
<td>2. STATE TABLE FOR $S(x) = x - x^2/2$</td>
<td>36</td>
</tr>
<tr>
<td>3. IMPLEMENTATION OF $S(x)$</td>
<td>36</td>
</tr>
<tr>
<td>4. CHEN'S TABLE OF PRIME IMPLICANTS FOR $p^2 = y + z$</td>
<td>42</td>
</tr>
<tr>
<td>5. TRUTH TABLE FOR THE GENERATION OF $z_9$</td>
<td>46</td>
</tr>
<tr>
<td>6. $z_9$ PRIME IMPLICANTS</td>
<td>47</td>
</tr>
<tr>
<td>7. PARTIAL TABLE OF BINARY LOGARITHMS</td>
<td>50</td>
</tr>
<tr>
<td>8. TABLE OF BINARY LOGARITHMS (STRAIGHT LINE APPROXIMATION)</td>
<td>53</td>
</tr>
</tbody>
</table>
INTRODUCTION

Modern day computers can spend a large amount of processing time performing time consuming operations like multiply and divide. The amount of time consumed on these operations depends primarily on the purpose for which the computer is used. When it is used for scientific computation and control, an arithmetic unit can spend nearly half of its processing time in the performance of multiplication and division.

The advent of bigger, larger, and faster computers has necessitated the corresponding development of larger and faster arithmetic units. Digital filtering and, more specifically, the fast Fourier transform have added to the impetus for faster arithmetic processing. As a result, much work has been done in the last ten years towards the development of more sophisticated and faster algorithms. The average engineer, in the author's experience, is totally unaware of these developments. Even if the engineer desires to design a better arithmetic unit and decides to research the field, the sheer bulk of material is enough to disuade him.

This report is an attempt to solve one portion of the above problem. It is an attempt to gather all the individual
papers on simultaneous multiplication techniques and to re-
view them in a single paper. This paper shall also serve as
an extensive bibliography on both multiplication and division
techniques. The first bibliography contains those books and
papers which pertain to the general theory of multiplication
and are not referenced in this paper. The second bibliography
is devoted solely to papers on division techniques. These
papers also result from the author's literature search.
NON-ITERATIVE MULTIPLY METHODS

As mentioned in the introduction, non-iterative multiply has been the subject of numerous papers in recent years. This is due mainly to the fact that, as modern day computers achieve greater and greater speeds, the multiply hardware must also become faster and faster. The non-iterative, or simultaneous, multiplier can be loosely defined as a multiplier which forms its product in one clock cycle. It is this speed which makes it attractive for use in high speed machines.

Consider two n-bit numbers, A and B, which are to be multiplied together. The multiplicand is \( A = A_nA_{n-1} \ldots A_1 \) and the multiplier is \( B = B_nB_{n-1} \ldots B_1 \). The addends can be represented as \( P_r \), where \( r = 1 \) to \( n \), so that:

\[
P_r = B_r \cdot A \quad \text{where} \quad \cdot \text{signifies logical AND}
\]

and \( B_r = 0 \) or 1, and \( P_r \) and A are n-bit numbers. The final product can be represented as:

\[
AB = Q = Q_{2n}Q_{2n-1} \ldots Q_1
\]

Using the relation in (1), Q can be written as:

\[
Q = P_1 + 2P_2 + 2^2P_3 + \ldots + 2^{n-1}P_n
\]

The interpretation of the relationship expressed in (3) is that the final product is the sum of the shifted addends, where each addend is the logical AND of a multiplier.
bit and the multiplicand \((A)\). Although equation \((3)\) can be implemented with iterative techniques, it can also be implemented as a non-iterative multiplier. This report shall discuss several algorithms which, in one form or another, implement equation \((3)\) as a simultaneous multiplier. The report will then proceed into the discussion of algorithms based on multiple-bit decoding, squaring, and binary logarithms. These also are distinguished by their ability to produce a product in one clock cycle.
PRIOSTE MULTIPLIER

One of the simplest non-iterative multipliers was discussed by Prioste \(^2, 3, 9\) of Motorola Semiconductor Products Inc. His approach consists basically of implementing equation (3) above. A block diagram of his approach is illustrated in Figure 1. The shifted version of each addend \((2^{r-1}P_r)\) is added in its own turn to an accumulating sum.

The total operation requires \(n-1\) total addition stages, where \(n\) is the number of bits in the multiplier. Each adder stage consists of a full multiplicand word adder. For a 16-bit by 16-bit multiply each adder must be capable of adding together two 16-bit numbers and there will be 15 stages of addition.

This method is highly applicable in systems which utilize medium and large scale integration (MSI and LSI). Current technology offers a 16-bit carry look-ahead adder consisting of five integrated circuits with a total add time of less than 50 nanoseconds. Using this adder configuration would result in a total multiply time of less than 750 ns. The fact that this implementation would require 75 integrated circuits, 60 of which are 24 pin devices makes it practical only in large systems where this number of devices does not represent a significant percentage of the logic count.
FIGURE 1

BLOCK DIAGRAM OF PRIOSTE MULTIPLIER
CARRY-SAVE MULTIPLIER

Another method for simultaneous multiplication which is very similar to the Prioste multiplier is a method implemented by the PCM Telemetry Laboratory of Purdue University. This approach is illustrated in Figure 3 utilizing the nomenclature of the 5-bit multiply in Figure 2. The basic philosophy behind this approach is that given an n-bit multiplicand \(a_n \cdots a_2a_1a_0\) and an m-bit multiplier \(b_m \cdots b_0\), the nm summands \(a_i b_j, i=1,2,\cdots n; j=1,2,\cdots m\), can be generated simultaneously by nm AND gates. Thus for the 5-bit case illustrated, 25, two input AND gates are needed to form the 25 summands. It can be seen that for a large nm multiplication the number of AND gates required to form the summands rises quite quickly. While it takes only 256 AND gates to form the summands for a 16 X 16-bit multiply, this number rises to 1024 AND gates for a 32 X 32-bit multiply.

Since the summands can be generated simultaneously, with only one level of gating, the problem reduces to that of reducing the time required to add the summands. The solution to this problem is two steps. In the first step, a set of two numbers whose sum is equal to the product is obtained from the reduction of the original set of summands. The second step consists of adding these two numbers to form the product.
FIGURE 2

MULTIPLICATION OF TWO 5-BIT NUMBERS
HA = HALF ADDER
FA = FULL ADDER

FIGURE 3

BLOCK DIAGRAM OF 5 X 5 MULTIPLIER USING CARRY-SAVE TECHNIQUE
In this approach n-1 stages are required to reduce the number of summands to the necessary two. Each stage is composed of n-1 adders. The first stage is composed entirely of half adders and the remaining n-2 stages are made up of full adders. The last stage which forms the product is composed of one half adder and n-2 full adders. The total number of adders required to form the entire product is n(n-1). Of this number n are half adders and the remaining n^2-2n adders are full adders.

This method offers little advantage over the Prioste multiplier if the Prioste multiplier utilizes carry look-ahead adders. The main difference in multiplication time will be whatever slight gate delays are accrued by the individual look-ahead circuits of each stage of addition in the Prioste multiplier. This difference might become sizable if the number of addition stages increases significantly, i.e. for extremely large n.

Since the last stage of the carry-save multiplier has no next stage which can handle its generated carries, it is best implemented with a carry look-ahead adder.

For a 16 X 16-bit multiply the carry-save multiplier will require 256 AND gates to form the summands, 15 half adders, 210 full adders, and one 15-bit carry look-ahead adder. Since the AND gates come four to a package and the adders, two to a package and the carry look-ahead adder...
represents 5 packages, a total of 178 14-pin packages and 4 24-pin packages would be required to implement this technique. Recalling that the package count for the Prioste multiplier was 15 14-pin packages and 60 24-pin packages, we are in a position to make a comparison. Utilizing the rule of thumb that a 24-pin package requires 2½ times the area as a 14-pin package, the package count for the Prioste multiplier becomes 165 chips while that of the carry-save multiplier becomes 191. Since the carry-save technique offers a real speed advantage, the choice reduces to whether the designer is willing to pay for this speed with extra hardware.
One of the earliest variations of the simultaneous multiplier was presented by Richards. His approach consists of grouping all the partial remainders, properly shifted, into pairs and adding all of these pairs simultaneously. Figure 4 illustrates the multiplication of two 8-bit numbers. In order to better demonstrate Richards' approach, each partial remainder is given the name W0, W1, through W7. The first stage of addition would group all the partial remainders into pairs and add all of them simultaneously. Thus R0 will be the result of the addition of W0 and W1, R1 will be formed from W2 and W3, R2 from W4 and W5, and R3 results from the addition of W6 and W7.

Still considering the 8 x 8 multiplication in Figure 4, the next step of Richards' approach is to group the results of the first stage addition into pairs and to add these simultaneously. Thus, Q0 is the result of the addition of R0 and R1 and Q1 results from R2 and R3.

The next stage of addition again groups the results of the previous stage into pairs and performs the addition. For the 8 x 8 multiplication, this addition is the terminal step. For the general case, the results of this addition will again be grouped into pairs and added. This process will continue until the number of terms to be added is reduced to
two. The final addition of these two terms forms the most significant portion of the product.
<table>
<thead>
<tr>
<th></th>
<th>a₇</th>
<th>a₆</th>
<th>a₅</th>
<th>a₄</th>
<th>a₃</th>
<th>a₂</th>
<th>a₁</th>
<th>a₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>b₇</td>
<td>b₆</td>
<td>b₅</td>
<td>b₄</td>
<td>b₃</td>
<td>b₂</td>
<td>b₁</td>
<td>b₀</td>
</tr>
<tr>
<td></td>
<td>a₇b₀</td>
<td>a₆b₀</td>
<td>a₅b₀</td>
<td>a₄b₀</td>
<td>a₃b₀</td>
<td>a₂b₀</td>
<td>a₁b₀</td>
<td>a₀b₀</td>
</tr>
<tr>
<td>R1</td>
<td>b₇</td>
<td>b₆</td>
<td>b₅</td>
<td>b₄</td>
<td>b₃</td>
<td>b₂</td>
<td>b₁</td>
<td>b₀</td>
</tr>
<tr>
<td></td>
<td>a₇b₁</td>
<td>a₆b₁</td>
<td>a₅b₁</td>
<td>a₄b₁</td>
<td>a₃b₁</td>
<td>a₂b₁</td>
<td>a₁b₁</td>
<td>a₀b₁</td>
</tr>
<tr>
<td>R2</td>
<td>b₇</td>
<td>b₆</td>
<td>b₅</td>
<td>b₄</td>
<td>b₃</td>
<td>b₂</td>
<td>b₁</td>
<td>b₀</td>
</tr>
<tr>
<td></td>
<td>a₇b₂</td>
<td>a₆b₂</td>
<td>a₅b₂</td>
<td>a₄b₂</td>
<td>a₃b₂</td>
<td>a₂b₂</td>
<td>a₁b₂</td>
<td>a₀b₂</td>
</tr>
<tr>
<td>R3</td>
<td>b₇</td>
<td>b₆</td>
<td>b₅</td>
<td>b₄</td>
<td>b₃</td>
<td>b₂</td>
<td>b₁</td>
<td>b₀</td>
</tr>
<tr>
<td></td>
<td>a₇b₃</td>
<td>a₆b₃</td>
<td>a₅b₃</td>
<td>a₄b₃</td>
<td>a₃b₃</td>
<td>a₂b₃</td>
<td>a₁b₃</td>
<td>a₀b₃</td>
</tr>
<tr>
<td></td>
<td>b₇</td>
<td>b₆</td>
<td>b₅</td>
<td>b₄</td>
<td>b₃</td>
<td>b₂</td>
<td>b₁</td>
<td>b₀</td>
</tr>
<tr>
<td></td>
<td>a₇b₄</td>
<td>a₆b₄</td>
<td>a₅b₄</td>
<td>a₄b₄</td>
<td>a₃b₄</td>
<td>a₂b₄</td>
<td>a₁b₄</td>
<td>a₀b₄</td>
</tr>
<tr>
<td>Q1</td>
<td>b₇</td>
<td>b₆</td>
<td>b₅</td>
<td>b₄</td>
<td>b₃</td>
<td>b₂</td>
<td>b₁</td>
<td>b₀</td>
</tr>
<tr>
<td></td>
<td>a₇b₅</td>
<td>a₆b₅</td>
<td>a₅b₅</td>
<td>a₄b₅</td>
<td>a₃b₅</td>
<td>a₂b₅</td>
<td>a₁b₅</td>
<td>a₀b₅</td>
</tr>
<tr>
<td>Q2</td>
<td>b₇</td>
<td>b₆</td>
<td>b₅</td>
<td>b₄</td>
<td>b₃</td>
<td>b₂</td>
<td>b₁</td>
<td>b₀</td>
</tr>
</tbody>
</table>
FIGURE 5

BLOCK DIAGRAM OF 8 X 8-BIT MULTIPLY USING RICHARDS' METHOD
Another scheme for the design of a simultaneous multiplier has been proposed by Dadda. It is unfortunate that the only description of this technique, available to the author, appears in a paper by Habibi and Wintz, which generates very little confidence in their results. The reason for this will be described in a subsequent treatment on the Wallace multiplier.

As in the carry-save technique, the Dadda multiplier relies on a two step approach to form the product. Considering, again, the 5 x 5-bit multiplication in Figure 2, the corresponding block diagram of the Dadda multiplier is illustrated in Figure 6. As in previous approaches, the various \( a_i b_j \) terms must be generated by a bank of AND gates. The two step approach for forming the product consists of, first, reducing the original set of summands to two numbers whose sum equals the product, and second, adding these two numbers to form the product.

The technique utilized by Dadda to perform the first step is to add each column in the summand matrix. As in the carry-save method, he allows the carries to propagate from the lower order to the higher order adders. An essential feature of the Dadda multiplier is that while the summand
Can be a string of full adders or a carry lookahead adder.

**FIGURE 6: BLOCK DIAGRAM OF 5 X 5 DADDA MULTIPLIER**
reduction is still taking place in the higher order adders on the left-hand side, the carry resulting from the lower order adders is propagating through the carry propagate adder. This results in a lower effective time delay.

The number of components in the Dadda multiplier is the same as the number used in the carry-save scheme. Since the approach is to add the columns of summands, we would expect the implementation to contain a higher density of adders in the center, where the column length is the greatest. The final stage of addition which adds the final two numbers to form the product can consist of a string of full adders, with ripple carry, or a faster carry look-ahead adder.
THE WALLACE MULTIPLIER

Wallace has proposed the use of carry-save adders, interconnected in a tree-like arrangement, to allow all additions to be done simultaneously. The tree network of full word adders has already been encountered in the Richards multiplier. The Wallace tree differs only in its use of full word carry-save adders.

The full word carry save adder is an adder in which three numbers are added to produce an output of two numbers whose sum is the sum of the original three numbers. The two output numbers are given the names SUM and CARRY. To better illustrate the operation of the full word carry-save adder, consider the following example and subsequent discussion,

```
1 0 0 1 0 1 1 0  INPUT WORD 1
1 1 1 0 1 1 1 0  INPUT WORD 2
0 0 0 1 0 0 1 1  INPUT WORD 3
0 1 1 0 1 0 1 1  SUM OUTPUT
1 0 0 1 0 1 1 0  CARRY OUTPUT
```

In the above example, the SUM output is obtained by the modulo 2 addition of each column of three bits. As a result, the sum will be a 0 if there are no 1's or if there are two 1's in the three bits to be added. The SUM output will be a 1 if there are an odd number of 1's in the three bits to
be added. The CARRY output is obtained by noting the carry which is generated by each addition of three bits. The CARRY is a 0 if there are one or less 1's in the three bits added. The CARRY is a 1 if there are two or more 1's in the three bits.

While the Richards multiplier tree groups the addends in groups of two and each adder stage reduces its two inputs to one output, in each addition stage; the Wallace multiplier tree groups the addends in groups of three and each adder reduces its three inputs to two outputs in each addition stage. Both approaches utilize as many stages as are required to reduce the original number of addends down to two. These final two numbers are added to form the product.

Since the Wallace multiplier reduces the addends by a factor of 1.5 at each addition stage and since the Richards multiplier reduces the number of addends by 2 at each addition stage, we can obtain a reasonable comparison of the number of stages required for each method if we consider a number whose word length is both a multiple of 3 and a multiple of 2.

For a 60 X 60 bit multiplication, we would expect the Wallace multiplier to utilize 40 \((60/1.5)\) addition stages while the Richards multiplier would use 30 \((60/2)\) addition stages. Since the Wallace multiplier groups the addends by
3's, it will have 20 adders, 61 bits long, in its first stage of addition. The Richards multiplier will have 30 adders, 60 bits long, in its first stage of addition since it groups the addends in pairs. This is illustrated, for \( n \) divisible by 6, in Figure 7, where the horizontal dimension represents the number of adders per stage and the vertical dimension represents the number of stages. The cases where the Richards multiplier must handle words not divisible by 2 or the cases where the Wallace multiplier must handle words not divisible by 3 will deviate from this graph.

In order to compensate for the additional stages required by his multiplication tree, Wallace introduces a scheme for pre-encoding the multiplier which reduces the number of addends by two. Before proceeding into this method, we can observe the effect on Figure 7 if the number of addends in the Wallace multiplier is cut in half while the number of addends in the Richards multiplier remains constant. This is illustrated in Figure 8. The effect on the previous 60 word example is to reduce it to a 30 word case for the Wallace multiplier, thereby reducing the number of required addition stages to 20 instead of 40.

The pre-encoding scheme, introduced by Wallace, selects each initial addend from a set of multiples of the multiplicand. The multiple used is determined by an overlapped
FIGURE 7. COMPARISON OF RICHARDS AND WALLACE TREE STRUCTURES
FIGURE 8. COMPARISON OF RICHARDS AND RECODED WALLACE TREE STRUCTURES
scan of three of the multiplier digits. The addend \( w_i \), where \( i \) is only odd, depends on the multiplier digits \( x_{i-1}, x_i \) and \( x_{i+1} \), where \( x_0 \) is the sign digit and largest \( i \) denotes the least significant bit. Digit \( x_{\text{LSB}+1} \) is taken as 0 if the largest \( i \) (signifying the LSB) is odd. The available multiples of the multiplicand are \(-2, -1, 0, 1, \) and 2. The rules of selection are as follows:

\[
\begin{align*}
-2 & \text{ if } \overline{x}_{i-1} \cdot x_i \cdot x_{i+1} \\
-1 & \text{ if } (\overline{x}_{i-1} \cdot x_i \cdot x_{i+1}) + (\overline{x}_{i-1} \cdot \overline{x}_i \cdot x_{i+1}) \\
0 & \text{ if } (\overline{x}_{i-1} \cdot \overline{x}_i \cdot x_{i+1}) + (x_{i-1} \cdot x_i \cdot x_{i+1}) \\
1 & \text{ if } (x_{i-1} \cdot x_i \cdot x_{i+1}) + (x_{i-1} \cdot \overline{x}_i \cdot x_{i+1}) \\
2 & \text{ if } x_{i-1} \cdot \overline{x}_i \cdot \overline{x}_{i+1}
\end{align*}
\]

where \( \cdot \) signifies logical AND

and \( + \) signifies logical OR

Representing the multiplicand as \( Y \), we can rewrite the above equations into the following table;

### TABLE 1

**ENCODING TABLE FOR WALLACE MULTIPLIER**

<table>
<thead>
<tr>
<th>( x_{i-1} )</th>
<th>( x_i )</th>
<th>( x_{i+1} )</th>
<th>ADDEND</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( Y )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>( Y )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>( 2Y )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>(-2Y )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>(-Y )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>(-Y )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Since \( i \) is only odd, the number of addends is reduced by a factor of two. The block diagram of the Wallace multiplier, for a 40 \( \times \) 40 bit multiplication, is illustrated in Figure 9. Each full word carry-save adder is numbered 1 to 18. The CARRY and SUM outputs are designated by \( c \) and \( s \) respectively in each box. Note that there are twenty addends entering the multiplier tree, which is one half the number of addends which enters the Richards multiplier.

To demonstrate the use of the Wallace multiplier, let us consider an example for the multiplication of two eight bit numbers. Let

\[
Y = \text{Multiplicand} = 0.0101010 = (42/128) \\
X = \text{Multiplier} = 0.0110110 = (54/128)
\]

For clarity, let us rewrite the multiplier in the following form:

\[
x_0 \ x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ x_7 \ (x_8) \\
0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ (0)
\]

The overlapped scan of three bits, which determines the multiples of the multiplicand, is also illustrated above. Using Table 1, we see that:

\[
W_1 = Y = 0.0101010 \\
W_3 = -Y = 1.1010110 \\
W_5 = 2Y = 0.1010100 \\
W_7 = -2Y = 1.0101100
\]
FIGURE 9. TREE STRUCTURE FOR 40 X 40 BIT WALLACE MULTIPLIER
Figure 10 illustrates the block diagram for the implementation of the above example. Since the addends are shifted relative to one another, the carry-save adder 1 must be twelve bits long while carry-save adder 2 is fourteen bits long. The process is illustrated arithmetically in figure 11.

Consider the similar block diagram for the 8 x 8 bit multiplication using the Richards method, in figure 5, and compare it with figure 10. The advantages are self-evident. The encoding of the multiplier, which reduces by two the number of addends to be handled, makes the Wallace multiplier attractive. However, Messrs. Habibi and Wintz, have failed to make any mention of this encoding technique in their treatment of Wallace's multiplier. As a result, their comparisons of various simultaneous multipliers is correct only if the Wallace multiplier is used without any pre-encoding. Since this is the most attractive element of the Wallace multiplier, the author feels that Messrs. Habibi and Wintz have thrown a shadow over their report by not including all aspects of the Wallace multiplier.
FIGURE 10.
BLOCK DIAGRAM FOR WALLACE MULTIPLIER EXAMPLE
<table>
<thead>
<tr>
<th>BITS</th>
<th>0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>W3</td>
<td>1 1 0 1 0 1 1 0</td>
</tr>
<tr>
<td>W5</td>
<td>0 0 0 1 0 1 0 1 0 0</td>
</tr>
<tr>
<td>W7</td>
<td>1 1 1 1 1 0 1 0 1 1 0 0</td>
</tr>
<tr>
<td>S1</td>
<td>0 0 0 0 1 1 1 0 0 0 1 1 0 0</td>
</tr>
<tr>
<td>C1</td>
<td>1 1 1 1 0 1 0 1 1 0 0 0 0 0 0</td>
</tr>
<tr>
<td>S1</td>
<td>0 0 0 0 1 1 1 0 0 0 1 1 1 0 0</td>
</tr>
<tr>
<td>SC1</td>
<td>1 1 1 0 1 0 1 1 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>W1</td>
<td>0 0 1 0 1 0 1 0 1 0</td>
</tr>
<tr>
<td>S2</td>
<td>1 1 0 0 .1 1 1 1 0 1 1 1 0 0</td>
</tr>
<tr>
<td>C2</td>
<td>0 0 1 0 1 0 1 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>S2</td>
<td>1 1 0 0 1 1 1 1 0 1 1 1 0 0</td>
</tr>
<tr>
<td>SC2</td>
<td>0 1 0 1 0 1 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

**PRODUCT** 0 0 1 0 0 0 1 1 0 1 1 0 0

**NOTE:** SC denotes shifted carry

**FIGURE 11**

**ARITHMETIC EXAMPLE OF WALLACE MULTIPLIER**
LING’S METHOD

In an attempt to reduce the large number of components necessary for simultaneous multipliers, Dr. Ling, of the IBM Research Laboratory, proposed a binary multiplication algorithm which is based on a multiple-bit decoding technique. His algorithm presupposes that the sum (I) and the difference (J) of the two numbers (A and B) to be multiplied have been computed in advance. This technique takes the original 2n-bit multiplication problem and transforms it into two well defined n-bit problems. By the use of his recoding method, Dr. Ling has succeeded in drastically reducing the number of adder steps required to perform a multiplication.

Consider two fractions, A and B, whose product is desired. If we define:

\[ I = \frac{(A + B)}{2} \quad (4) \]
\[ J = \frac{(A - B)}{2} \quad (5) \]

and

\[ f(x) = \frac{(x)}{2}(x + 1) \quad (6) \]

Then the product AB can be written as:

\[ AB = 2 \left[ f(I) - f(J) - \frac{(I-J)}{2} \right] \quad (7) \]

For this solution to have any usefulness, it is necessary to decompose \( f(I) \) and \( f(J) \) into forms which are
more manageable. In order to accomplish this, three lemmas are established which will aid in the decomposition.

Lemma 1. For $d = 0$ or $1$, $f(d) = d$  \hspace{1cm} (8)

Lemma 2. $f(x + y) = f(x) + f(y) + xy$  \hspace{1cm} (9)

Lemma 3. $f(ny) = n^2 f(y) - yf(n-1)$  \hspace{1cm} (10)

For $n = 2$ Lemma 3. can be written as:

$$f(2y) = 4f(y) - yf(1)$$  \hspace{1cm} (11)

Observing that $f(1) = 1$, (11) can be rewritten to give:

$$f(y) = \frac{1}{4}y + \frac{3}{4}f(2y)$$  \hspace{1cm} (12)

Taking a binary fraction of the form

$$x = \sum_{n=1}^{k} x_n 2^{-n} \quad \text{where } x_n = 0 \text{ or } 1$$  \hspace{1cm} (13)

and equating it to $y$ in equation (12), yields:

$$f(0, x_1 x_2 x_3 \ldots x_n) = \frac{1}{4}(0, x_1 x_2 x_3 \ldots x_n)$$

$$+ \frac{3}{4}f(x_1, x_2 x_3 \ldots x_n)$$  \hspace{1cm} (14)

By using Lemma 2, we can express the second part of equation (14) as;

$$f(x_1, x_2 x_3 \ldots x_n) = f(x_1) + f(0, x_2 x_3 \ldots x_n)$$

$$+ x_1 (0, x_2 x_3 \ldots x_n)$$  \hspace{1cm} (15)

Using the fact that a binary element is its own square, we can express the first and third part of equation (15) as;

$$f(x_1) + x_1 (0, x_2 x_3 \ldots x_n) = \frac{x_1}{2} (x_1 + 1) + x_1 (0, x_2 \ldots x_n)$$

Adding zero in the form of $x_1 \bar{x}_1/2$, we have
\[ f(x_1) + x_1(0.x_2 \ldots x_n) = x_1 \left[ \left( x_1/2 \right) + \left( x_2/2 \right) + \left( x_1/2 \right) \right. \]
\[ + \left. (0.x_2 x_3 \ldots x_n) \right] \]
\[ = (x_1 x_1/2) + x_1(x_1 x_2 \ldots x_n) \]
\[ = 0 + 2x_1(0.x_1 x_2 x_3 \ldots x_n) \] (16)

Substituting (15) into (16) yields:
\[ f(x_1 x_2 x_3 \ldots x_n) = 2x_1(0.x_1 x_2 x_3 \ldots x_n) \]
\[ + f(0.x_2 x_3 \ldots x_n) \] (17)

Substituting (17) back into (15), we have
\[ f(0.x_1 x_2 x_3 \ldots x_n) = \frac{1}{4} (0.x_1 x_2 x_3 \ldots x_n) \]
\[ + \frac{2}{4} x_1(0.x_1 x_2 x_3 \ldots x_n) \]
\[ + \frac{1}{4} f(0.x_2 x_3 \ldots x_n) \] (18)

Repeating the above decomposition n times yields:
\[ f(0.x_1 \ldots x_n) = \frac{1}{4} (0.x_1 \ldots x_n) + \frac{2}{4} x_1(0.x_1 \ldots x_n) \]
\[ + \frac{1}{4} x_2(0.x_2 \ldots x_n) + \frac{2}{4} x_3(0.x_3 \ldots x_n) \]
\[ + \frac{1}{4} x_4(0.x_4 \ldots x_n) + \frac{2}{4} x_n(0.x_n) \]
\[ + \frac{1}{4} x_{n+1}(0.x_{n+1} \ldots x_n) \]
\[ + \frac{1}{4} x_{n+2}(0.x_{n+2} \ldots x_n) \]
\[ + \cdots \]
\[ + \frac{1}{4} x_{2n}(0.x_{2n} \ldots x_n) \]
\[ + \frac{1}{4} x_{2n+1}(0.x_{2n+1} \ldots x_n) \]
\[ + \frac{1}{4} x_{2n+2}(0.x_{2n+2} \ldots x_n) \]
\[ + \cdots \]
\[ + \frac{1}{4} x_{4n}(0.x_{4n} \ldots x_n) \]
\[ + \frac{1}{4} x_{4n+1}(0.x_{4n+1} \ldots x_n) \]
\[ + \cdots \]
\[ + \frac{1}{4} x_{4n+k}(0.x_{4n+k} \ldots x_n) \] (19)

Equation (19) can be rewritten in the form
\[ f(0.x_1 \ldots x_n) = \sum_{k=1}^{n} \frac{1}{4^k} (1 + 2x_k) \sum_{m=1}^{n-k+1} 2^{-m} x_{m+k-1} \]
By letting
\[ D(x) = \sum_{k=1}^{n} \frac{1}{4^k} \sum_{m=1}^{n-k+1} 2^{-m} x_{m+k-1} \]
and
\[ L(x) = \sum_{m=1}^{n} \frac{1}{4^m} x_{m} \sum_{k=1}^{n-k+1} 2^{-m} x_{m+k-1} \]
we obtain
\[ f(x) = D(x) + 2L(x) \quad (20) \]
where \( D(x) \) and \( L(x) \) can be expressed as;
\[ D(x) = (1/2)x - B(x) \]
where
\[ B(x) = 0.0 \quad 0x_1 \quad 0x_2 \quad 0x_3 \quad 0x_4 \quad \ldots \quad 0x_{n-1} \quad 0x_n \]
and
\[ L(x) = 0.0 \quad 0 \quad (x_1x_1) \quad (x_1x_2) \quad (x_1x_3) \quad \ldots \quad (x_1x_n) \]
\[ + 0.0 \quad 0 \quad 0 \quad 0 \quad (x_2x_2) \quad \ldots \quad (x_2x_n) \]
\[ \vdots \]
\[ + 0.0 \quad 0 \quad 0 \quad 0 \quad 0 \quad \ldots \quad (x_nx_n) \quad (21) \]
Dr. Ling then manipulates equation (20), in a manner which the author has not yet duplicated, to form;
\[ f(x) = 3D(x) - 2\phi(x) \quad (22) \]
where;
\[ \phi(x) = 0.0 \quad 0 \quad (\bar{x}_1x_1) \quad (\bar{x}_1x_2) \quad (\bar{x}_1x_3) \quad \ldots \quad (\bar{x}_1x_n) \]
\[ + 0.0 \quad 0 \quad 0 \quad 0 \quad (\bar{x}_2x_2) \quad \ldots \quad (\bar{x}_2x_n) \]
\[ \vdots \]
\[ + 0.0 \quad 0 \quad 0 \quad 0 \quad 0 \quad \ldots \quad (\bar{x}_{n-1}x_n) \quad (23) \]
Since we have limited our treatment, thusfar, to binary fractions, it is desirable to extend the theory so that this
approach is applicable to integers.

If I and J are both binary integers;

\[ I = 2^n(0.i_1 i_2 i_3 \ldots i_n) = 2^ni \]  
(24)

\[ J = 2^m(0.j_1 j_2 j_3 \ldots j_n) = 2^mj \]  
(25)

and equation (7) can be rewritten as

\[ AB = 2\{ f[2^n i] - f[2^m j] - (I - J)/2 \} \]  
(26)

Using Lemma 3, we can express (26) as

\[ AB = 2\{2^n f(i) - (2^n - 1)I/2 - 2^m f(j) - (2^m - 1)J/2 \} \]  
-(I - J) \} \]  
(27)

which can be simplified to

\[ AB = 2\{2^n f(i) - 2^m f(j) - 2^n\left(\frac{1}{2}\right) + 2^m\left(\frac{j}{2}\right) \} \]  
(28)

Substuting (22) into (28) yields;

\[ AB = 2\{2^n [3B(i) - 2\varphi(i)] - 2^m [3B(j) - 2\varphi(j)] \} \]  
- I \left(\frac{n}{2}\right) + J \left(\frac{m}{2}\right) \} \]  
(29)

Using the relation \( B(x) = (1/2)x - D(x) \) we have;

\[ AB = 2\{2^n I - 2^m J - 2^n [3B(i) + 2\varphi(i)] + 2^m [3B(j) + 2\varphi(j)] \} \]

Letting \( S(x) = 3B(x) + 2\varphi(x) \) yields

\[ AB = 2\{2^n I - 2^m J - 2^n [S(i)] + 2^m [S(j)] \} \]  
(30)

Equation (30) implies that, given a combinatorial logic box which will form the function \( S(x) = x - (x^2/2) \), the multiplication of two numbers can be accomplished with four stages of addition. This offers a significant improvement over the simultaneous multipliers discussed thusfar.
In his paper, Dr. Ling presents a table of prime implicants required for the generation of the $S(x)$ transfer function. Dr. Ling seems to feel that any word size larger than eight bits will not justify the amount of hardware necessary to implement the $S(x)$ function. He recommends that any number larger than eight bits be decomposed into eight bit blocks and increasing the number of addition stages.

For purposes of this paper, the author feels that it is adequate to consider the multiplication of two four bit words in order to demonstrate Dr. Ling's method. The first necessary step is to generate the $S(x)$ transfer box. This exercise is simplified by the restriction $1 > x \geq \frac{1}{2}$, which implies that $x_1$ equals 1. The state table for $S(x) = x - x^2/2$ is illustrated in Table 2, where the inputs are $0, x_1, x_2, x_3, x_4$ and the outputs are $0, S_0, S_1, \ldots, S_7$. Table 3 illustrates the logical terms which are utilized to form each $S_i$ bit.

Consider two numbers, $A = 14$ and $B = 10$, whose binary representation is;

\[
\begin{align*}
A &= 14_{10} = 1110_2 \\
B &= 10_{10} = 1010_2
\end{align*}
\]

from (1), (2), (21) and (22)

\[
\begin{align*}
I &= 2^4(.1100) \\
J &= 2^2(.10)
\end{align*}
\]

Substituting $I$ and $J$ into Table 3 yields
TABLE 2

STATE TABLE FOR S(x) \( x - x^2/2 \)

<table>
<thead>
<tr>
<th>( S_0 )</th>
<th>( S_1 )</th>
<th>( S_2 )</th>
<th>( S_3 )</th>
<th>( S_4 )</th>
<th>( S_5 )</th>
<th>( S_6 )</th>
<th>( S_7 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>.1000</td>
<td>0.0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>.1001</td>
<td>0.0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>.1010</td>
<td>0.0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>.1011</td>
<td>0.0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>.1100</td>
<td>0.0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>.1101</td>
<td>0.0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>.1110</td>
<td>0.0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>.1111</td>
<td>0.0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

TABLE 3

IMPLEMENTATION OF S(x)

\( S_0 = x_1 \)
\( S_1 = x_1 \)
\( S_2 = x_1x_2 + x_1x_3x_4 \)
\( S_3 = x_1x_2 + x_1x_3\overline{x}_4 \)
\( S_4 = x_1\overline{x}_2\overline{x}_3x_4 + x_1x_3x_4 + x_1x_2x_3 \)
\( S_5 = x_1x_3 + x_1x_4 \)
\( S_6 = x_1x_4 \)
\( S_7 = x_1x_4 \)
The product is now formed by applying (27) in the following manner, with three additions:

\[
AB = 2 \left[ 0.11000000 \\
- 1000 \\
- 01110000 \\
+ 0110 \right]
\]

ANS. \( 10001100_2 = 140_{10} \)
CHEN'S METHOD

Shortly after the appearance of Ling's paper on the multiple bit decoding algorithm, Tien Chi Chen of IBM's San Jose Research Laboratory submitted a paper on a multiplication algorithm based on squaring which is very similar to Dr. Ling's. Dr. Chen noted the similarity of Ling's method to the quarter square multiplication method used in analog computation. This method is based on the following relation:

\[ A \cdot B = \left( \frac{A + B}{2} \right)^2 - \left( \frac{A - B}{2} \right)^2 \]

Dr. Chen felt that a multiplication scheme based on this relation could result in a more efficient implementation resulting in less hardware.

The basic problem essentially consists in decomposing the square of an arbitrary number into the sum of two or three terms with minimum logic complexity. While Chen limits his treatment to fractional numbers, the results are applicable to integers. Decomposing a square into the sum of two or three terms implies that the following relation must be satisfied:

\[ P^2 = Y + Z = R + S + T \]

The "multiplication parallelogram" for two 8-bit numbers is illustrated in Figure 12, with a typical term being \( P_{ij} = P_i \cdot P_j \), the dot signifying logical AND. By
FIGURE 12

8 X 8-BIT MULTIPLICATION PARALLELOGRAM
exploiting the symmetry of the partial product matrix it is possible to express the square as the sum of five terms as follows:

\[ p^2 = J + K + L + M + N \]

where \( J, K, L, M \), and \( N \) consist of the following binary words:

\[
J = 0.0 \ p_1 \ 0 \ p_2 \ 0 \ p_3 \ 0 \ p_4 \ 0 \ p_5 \ 0 \ p_6 \ 0 \ p_7 \ 0 \ p_8 \\
K = 0.0 \ p_{12} \ p_{13} \ p_{14} \ p_{15} \ p_{16} \ p_{17} \ p_{18} \ p_{28} \ p_{38} \ p_{48} \ p_{58} \ p_{68} \ p_{78} \\
L = 0.0 \ 0 \ 0 \ 0 \ p_{23} \ p_{24} \ p_{25} \ p_{26} \ p_{27} \ p_{37} \ p_{47} \ p_{57} \ p_{67} \\
M = 0.0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ p_{34} \ p_{35} \ p_{36} \ p_{46} \ p_{56} \\
N = 0.0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ p_{45}
\]

It is possible to remove the dependency of \( J \) on \( p_1, p_2, p_7, \) and \( p_8 \) by compensating \( K \) and \( L \). Note that the sum \( J + K + L \) remains unchanged if the \( 0 \) and \( p_8 \) terms on the right-hand side are removed from \( J \) and appended to the right-hand side of \( K \). Similarly, the sum is unchanged if the \( 0 \) and \( p_7 \) terms of \( J \) are moved to the corresponding positions of \( L \). This now removes the dependency of \( J \) on \( p_1 \) and \( p_8 \). In order to remove the dependency of \( J \) on \( p_1 \) and \( p_2 \) consider the following truth table for the function \( F = (f_1 f_2) = p_i \) plus \( p_{ij} \).
\[
\begin{array}{cccc}
P_i & P_j & f_1 & f_2 \\
0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 \\
\end{array}
\]

Therefore \( p_1 \) plus \( p_{ij} = p_{ij} \cdot (p_iP_j) \). Thus the sum of \( p_1 \) and \( p_{12} \) can be preserved by eliminating \( p_1 \) from \( J \), and changing \( 0 p_{12} \) into the two terms \( p_{12} \) and \( (p_1P_2) \) in \( K \). Similarly, \( p_2 \) and \( p_{23} \) can be removed from \( J \) and \( L \) respectively and replaced by \( p_{23} \) and \( (p_2P_3) \). The resulting terms \( J^* \), \( K^* \), and \( L^* \) preserve the sum of \( J \), \( K \), and \( L \) with the added feature that \( J^* \) is no longer dependent on \( p_1 \), \( p_2 \), \( p_7 \), and \( p_8 \). The resulting terms can be written as:

\[
J^* = 0.0 0 0 0 0 0 p_3 0 p_4 0 p_5 0 p_6
\]

\[
K^* = 0.0 p_{12}(p_1P_2) p_{13} p_{14} p_{15} p_{16} p_{17} p_{18} p_{28} p_{38} p_{48} p_{58} p_{68} p_{78} 0 p_8
\]

\[
L^* = 0.0 0 p_{23}(p_2P_3) p_{24} p_{25} p_{26} p_{27} p_{37} p_{47} p_{57} p_{67} 0 p_7
\]

and \( J + K + L = J^* + K^* + L^* \).

By identifying \( Y = K^* \) and \( Z = J^* + L^* + M + N \) the previously sought relation, \( p = Y + Z \), is achieved. Note that \( Z \) involves only the variables \( p_2 \) through \( p_7 \). The analysis of the expression for \( Z \) results in a table of prime implicants illustrated in Table 4, which is taken directly from Chen's paper. It should be noted that, in the case of an \( 8 \times 8 \)-bit multiplication, each \( z \) term involves no more than 14 prime implicants and each bit involves no more than
TABLE 4
CHEN’S TABLE OF PRIME IMPLICANTS FOR $p^2 = y + z$

\[
Y = 0. p_{12}(p_{1} \overline{p_2})p_{13}p_{14}p_{15}p_{16}p_{17}p_{18}p_{28}p_{38}p_{48}p_{58}p_{68}p_{78}0p_8 = K^*
\]
\[
Z = 0.0.0 z_3z_4z_5z_6z_7z_8z_9z_{10}z_11z_12z_13z_{14} = J^* + L^* + M + N
\]

<table>
<thead>
<tr>
<th>$p_2$</th>
<th>$p_3$</th>
<th>$p_4$</th>
<th>$p_5$</th>
<th>$p_6$</th>
<th>$p_7$</th>
<th>$p_2$</th>
<th>$p_3$</th>
<th>$p_4$</th>
<th>$p_5$</th>
<th>$p_6$</th>
<th>$p_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$z_3$</td>
<td>$1$</td>
<td>$0$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$x$</td>
<td>$1$</td>
<td>$0$</td>
<td>$0$</td>
<td>$x$</td>
<td>$x$</td>
</tr>
<tr>
<td></td>
<td>$1$</td>
<td>$1$</td>
<td>$x$</td>
<td>$x$</td>
<td>$x$</td>
<td>$x$</td>
<td>$1$</td>
<td>$0$</td>
<td>$1$</td>
<td>$0$</td>
<td>$x$</td>
</tr>
<tr>
<td></td>
<td>$1$</td>
<td>$0$</td>
<td>$1$</td>
<td>$0$</td>
<td>$x$</td>
<td>$x$</td>
<td>$1$</td>
<td>$0$</td>
<td>$0$</td>
<td>$x$</td>
<td>$x$</td>
</tr>
<tr>
<td>$z_5$</td>
<td>$0$</td>
<td>$1$</td>
<td>$0$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$z_4$</td>
<td>$1$</td>
<td>$0$</td>
<td>$0$</td>
<td>$x$</td>
</tr>
<tr>
<td></td>
<td>$0$</td>
<td>$1$</td>
<td>$1$</td>
<td>$x$</td>
<td>$x$</td>
<td>$x$</td>
<td></td>
<td>$0$</td>
<td>$1$</td>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td></td>
<td>$1$</td>
<td>$0$</td>
<td>$1$</td>
<td>$0$</td>
<td>$x$</td>
<td>$x$</td>
<td></td>
<td>$0$</td>
<td>$1$</td>
<td>$0$</td>
<td>$1$</td>
</tr>
<tr>
<td></td>
<td>$1$</td>
<td>$0$</td>
<td>$1$</td>
<td>$1$</td>
<td>$0$</td>
<td>$x$</td>
<td></td>
<td>$0$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td></td>
<td>$1$</td>
<td>$1$</td>
<td>$x$</td>
<td>$1$</td>
<td>$x$</td>
<td>$x$</td>
<td></td>
<td>$1$</td>
<td>$0$</td>
<td>$0$</td>
<td>$1$</td>
</tr>
<tr>
<td></td>
<td>$1$</td>
<td>$1$</td>
<td>$0$</td>
<td>$0$</td>
<td>$1$</td>
<td>$1$</td>
<td></td>
<td>$1$</td>
<td>$0$</td>
<td>$1$</td>
<td>$0$</td>
</tr>
<tr>
<td></td>
<td>$1$</td>
<td>$1$</td>
<td>$0$</td>
<td>$1$</td>
<td>$0$</td>
<td>$x$</td>
<td></td>
<td>$1$</td>
<td>$1$</td>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td></td>
<td>$1$</td>
<td>$1$</td>
<td>$0$</td>
<td>$0$</td>
<td>$1$</td>
<td>$1$</td>
<td></td>
<td>$1$</td>
<td>$1$</td>
<td>$0$</td>
<td>$1$</td>
</tr>
<tr>
<td></td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$x$</td>
<td>$1$</td>
<td>$x$</td>
<td></td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$x$</td>
</tr>
<tr>
<td>( p_2 )</td>
<td>( p_3 )</td>
<td>( p_4 )</td>
<td>( p_5 )</td>
<td>( p_6 )</td>
<td>( p_7 )</td>
<td>( p_2 )</td>
<td>( p_3 )</td>
<td>( p_4 )</td>
<td>( p_5 )</td>
<td>( p_6 )</td>
<td>( p_7 )</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>( z_7 = X ; 1 ; 0 ; 1 ; 0 ; X )</td>
<td>( z_8 = 0 ; 0 ; 1 ; 0 ; X ; X )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 1 X X</td>
<td>0 0 1 1 1 X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 1 1 0</td>
<td>0 1 0 0 1 X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 X 1 X</td>
<td>0 1 0 1 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 X 1 X</td>
<td>0 1 1 0 0 X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1 0 1 0</td>
<td>0 1 1 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1 1 0 X</td>
<td>1 0 0 X X 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1 1 1 1</td>
<td>1 0 1 0 X 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0 0 1 0</td>
<td>1 0 1 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0 1 1 1</td>
<td>1 1 0 X 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 X X 1</td>
<td>1 1 0 X 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1 0 0 1</td>
<td>1 1 0 1 1 X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 0 0 0</td>
<td>0 1 1 1 X 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### TABLE 4. (CONTINUED)

<table>
<thead>
<tr>
<th></th>
<th>( p_2 )</th>
<th>( p_3 )</th>
<th>( p_4 )</th>
<th>( p_5 )</th>
<th>( p_6 )</th>
<th>( p_7 )</th>
<th>( p_2 )</th>
<th>( p_3 )</th>
<th>( p_4 )</th>
<th>( p_5 )</th>
<th>( p_6 )</th>
<th>( p_7 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( z_9 )</td>
<td>( \times \times 0 1 1 0 )</td>
<td>( z_{10} )</td>
<td>( \times \times 0 1 0 \times )</td>
<td>( \times \times 1 0 1 0 )</td>
<td>( \times \times 0 1 1 1 )</td>
<td>( \times \times 0 1 1 1 )</td>
<td>( \times \times 1 1 0 0 )</td>
<td>( \times \times 1 1 0 0 )</td>
<td>( \times \times 1 1 0 0 )</td>
<td>( \times \times 1 1 0 0 )</td>
<td>( \times \times 1 1 0 0 )</td>
<td>( \times \times 1 1 0 0 )</td>
</tr>
<tr>
<td>( z_{11} )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 )</td>
<td>( \times \times 0 1 )</td>
<td>( \times \times 0 1 )</td>
<td>( \times \times 0 1 )</td>
<td>( \times \times 0 1 )</td>
<td>( \times \times 0 1 )</td>
<td>( \times \times 0 1 )</td>
<td>( \times \times 0 1 )</td>
<td>( \times \times 0 1 )</td>
<td>( \times \times 0 1 )</td>
<td>( \times \times 0 1 )</td>
</tr>
<tr>
<td>( z_{12} )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
</tr>
<tr>
<td>( z_{13} )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
</tr>
<tr>
<td>( z_{14} )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
<td>( \times \times 0 1 1 )</td>
</tr>
</tbody>
</table>
6 variables, which compares with 26 and 7 respectively for the Ling approach. The author recommends caution when using the table presented by Chen directly in as much as one error and one omission has been discovered by the author.

Before illustrating Chen's method with an example, it is advisable to review the corrections made to Chen's table and the analysis undergone to arrive at these corrections since the results are necessary for the example.

The error was discovered in the prime implicants necessary for the generation of \( z_9 \). In order to ascertain the correct terms, a truth table must be generated for the following addition:

\[
\begin{array}{cccccc}
0 & p_5 & 0 & p_6 \\
p_{37} & p_{47} & p_{57} & p_{67} & 0 & p_7 \\
p_{46} & p_{56} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

Note that the above addition involves only the variables \( p_3, p_4, p_5, p_6, \) and \( p_7 \). The truth table for the generation of \( z_9 \) is illustrated in Table 5.

Expressing \( z_9 \) as the sum of minterms yields:
TABLE 5

TRUTH TABLE FOR THE GENERATION OF \( z_9 \)

<table>
<thead>
<tr>
<th>3 4 5 6 7</th>
<th>9 10 11 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>0 0 0 0 1</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>0 0 0 1 0</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>0 0 1 0 0</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>0 0 1 0 1</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>0 0 1 1 0</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>0 0 1 1 1</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>0 1 0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>0 1 0 0 1</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>0 1 0 1 0</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>0 1 0 1 1</td>
<td>1 1 1 0</td>
</tr>
<tr>
<td>0 1 1 0 0</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>0 1 1 0 1</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>0 1 1 1 0</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>0 1 1 1 1</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>1 0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1 0 0 0 1</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>1 0 0 1 0</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>1 0 0 1 1</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>1 0 1 0 0</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>1 0 1 0 1</td>
<td>1 1 1 0</td>
</tr>
<tr>
<td>1 0 1 1 0</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>1 0 1 1 1</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>1 1 0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1 1 0 0 1</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>1 1 0 1 0</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>1 1 0 1 1</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>1 1 1 0 0</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>1 1 1 0 1</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>1 1 1 1 0</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>1 1 1 1 1</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>
which will reduce to the following set of prime implicants:

\[ z_9 = P_3\bar{P}_4P_5\bar{P}_6P_7 + P_3P_4P_5\bar{P}_6P_7 + P_3\bar{P}_4P_5P_6P_7 + \]
\[ P_3P_4\bar{P}_5P_6P_7 + P_3\bar{P}_4\bar{P}_5P_6P_7 + P_3\bar{P}_4P_5\bar{P}_6P_7 + \]
\[ P_3\bar{P}_4\bar{P}_5P_6P_7 + P_3\bar{P}_4\bar{P}_5\bar{P}_6P_7 + P_3\bar{P}_4\bar{P}_5\bar{P}_6P_7 + \]
\[ P_3\bar{P}_4\bar{P}_5\bar{P}_6P_7 + P_3\bar{P}_4\bar{P}_5\bar{P}_6P_7 + \]

Tabularizing the results in the form that Chen utilizes gives:

\[
\begin{array}{cccccc}
\text{z}_9 & \text{PRIME IMPLICANTS} \\
\hline
P_2 & P_3 & P_4 & P_5 & P_6 & P_7 \\
X & X & 0 & 1 & 1 & 0 \\
X & X & 1 & 0 & 1 & 0 \\
X & 0 & 1 & 0 & 1 & 1 \\
X & 0 & 1 & 1 & 0 & 1 \\
X & 0 & X & 1 & 1 & 1 \\
X & 1 & 0 & 0 & 1 & 1 \\
X & 1 & 0 & 1 & 0 & 1 \\
X & 1 & 1 & 0 & 0 & 1 \\
X & 1 & 0 & 0 & 0 & 1 \\
\end{array}
\]

After replacing the above terms for those in Chen's table we are ready to consider an example. Consider two binary numbers A and B where:

\[
A = 0,11110000.
\]
\[
B = 0,11010100
\]
\[
\frac{A + B}{2} = 0.11100010
\]
\[
\frac{A - B}{2} = 0.0001110
\]
\[
\left( \frac{A + B}{2} \right)^2 = K* + Z \quad K* = 0.1010001000000000
\]
\[
Z = 0.00100101100001
\]
\[
K* + Z = 0.11000111100001
\]
\[
\left( \frac{A - B}{2} \right)^2 = K* + Z \quad K* = 0.0000000000000000
\]
\[
Z = 0.00000001100010
\]
\[
K* + Z = 0.00000001100010
\]

Since \( AB = \left( \frac{A + B}{2} \right)^2 - \left( \frac{A - B}{2} \right)^2 \)

\[
AB = 0.11000111100001
\]
\[
1.1111111101111100
\]
\[
0.1100011011000000
\]

which is the correct result.
MITCHELL'S METHOD

Mitchell has proposed a method for binary multiplication which will reduce the process to a simple add or subtract and shift through the use of binary logarithms. Logarithms have long been used as a tool in mathematics to reduce multiplication to a simple add process. Its use in computing machines is prohibitive due to large storage requirement for the logarithms. On the other hand the calculation of the logarithms would consume more time than the multiplication process itself. Mitchell's method is to present a set of approximations to the binary logarithms. Since these approximations will introduce errors in the result, he also presents a method for reducing the final error.

Define \( \lg N = \log_2 N \) in order to avoid the necessity of writing the subscript. A table of binary logarithms is illustrated in Table 7, and the familiar logarithmic curve is plotted in Figure 13. An approximation of the binary logarithm can be generated by connecting all the points where \( \lg N \) is an integer with straight lines. This approximation is shown in Figure 14, with the results illustrated in Table 8. The characteristic of the logarithm is the exponent of the most significant "one" bit.

The approximate mantissa consists of the remaining bits to the right, which are interpreted as a binary fraction.
### TABLE 7

**PARTIAL TABLE OF BINARY LOGARITHMS**

<table>
<thead>
<tr>
<th>( N )</th>
<th>( \log N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00000</td>
</tr>
<tr>
<td>2</td>
<td>1.00000</td>
</tr>
<tr>
<td>3</td>
<td>1.58496</td>
</tr>
<tr>
<td>4</td>
<td>2.00000</td>
</tr>
<tr>
<td>5</td>
<td>2.32193</td>
</tr>
<tr>
<td>6</td>
<td>2.58496</td>
</tr>
<tr>
<td>7</td>
<td>2.80735</td>
</tr>
<tr>
<td>8</td>
<td>3.00000</td>
</tr>
<tr>
<td>9</td>
<td>3.16992</td>
</tr>
<tr>
<td>10</td>
<td>3.32193</td>
</tr>
<tr>
<td>11</td>
<td>3.45942</td>
</tr>
<tr>
<td>12</td>
<td>3.58496</td>
</tr>
<tr>
<td>13</td>
<td>3.70043</td>
</tr>
<tr>
<td>14</td>
<td>3.80735</td>
</tr>
<tr>
<td>15</td>
<td>3.90689</td>
</tr>
<tr>
<td>16</td>
<td>4.00000</td>
</tr>
<tr>
<td>17</td>
<td>4.08747</td>
</tr>
</tbody>
</table>
FIGURE 13. BASE 2 LOGARITHM CURVE
FIGURE 14. BASE 2 LOGARITHM CURVE (APPROXIMATION)
### TABLE 8

**TABLE OF BINARY LOGARITHMS**

*(STRAIGHT-LINE APPROXIMATION)*

<table>
<thead>
<tr>
<th>N</th>
<th>N (binary)</th>
<th>APPROX. lg N</th>
<th>APPROX. lg N (binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00001</td>
<td>0.000</td>
<td>000.0000</td>
</tr>
<tr>
<td>2</td>
<td>00010</td>
<td>1.000</td>
<td>001.0000</td>
</tr>
<tr>
<td>3</td>
<td>00011</td>
<td>1.500</td>
<td>001.1000</td>
</tr>
<tr>
<td>4</td>
<td>00100</td>
<td>2.000</td>
<td>010.0000</td>
</tr>
<tr>
<td>5</td>
<td>00101</td>
<td>2.250</td>
<td>010.0100</td>
</tr>
<tr>
<td>6</td>
<td>00110</td>
<td>2.500</td>
<td>010.1000</td>
</tr>
<tr>
<td>7</td>
<td>00111</td>
<td>2.750</td>
<td>010.1100</td>
</tr>
<tr>
<td>8</td>
<td>01000</td>
<td>3.000</td>
<td>011.0000</td>
</tr>
<tr>
<td>9</td>
<td>01001</td>
<td>3.125</td>
<td>011.0010</td>
</tr>
<tr>
<td>10</td>
<td>01010</td>
<td>3.250</td>
<td>011.0100</td>
</tr>
<tr>
<td>11</td>
<td>01011</td>
<td>3.375</td>
<td>011.0110</td>
</tr>
<tr>
<td>12</td>
<td>01100</td>
<td>3.500</td>
<td>011.1000</td>
</tr>
<tr>
<td>13</td>
<td>01101</td>
<td>3.625</td>
<td>011.1010</td>
</tr>
<tr>
<td>14</td>
<td>01110</td>
<td>3.750</td>
<td>011.1100</td>
</tr>
<tr>
<td>15</td>
<td>01111</td>
<td>3.875</td>
<td>011.1110</td>
</tr>
<tr>
<td>16</td>
<td>10000</td>
<td>4.000</td>
<td>100.0000</td>
</tr>
<tr>
<td>17</td>
<td>10001</td>
<td>4.0625</td>
<td>100.0010</td>
</tr>
</tbody>
</table>
within the range 0 to 1. The reasoning for this can be illustrated by considering the approximate logarithm curve in Figure 14. Each straight line segment in the curve has its start and end at some power of two. These exponents are the characteristics. Each line segment represents the transition from one characteristic to another which is only 1 higher in value. Thus the line segment represents the transition from 0 to 1.

In order to find the binary logarithm of a number, record the bit position of the most significant "one" and interpret the rest of the number as a binary fraction. As an example, consider the problem of finding the approximate \( \log_2 13 \) which, from Table 8 should be 3.625. The binary representation of 13 is 1101. The most significant "one" bit is in position \( 2^3 \) which means that the characteristic is three. The remaining bits result in a fraction 0.101 which is equivalent to .625 in decimal notation. Thus the approximate \( \log_2 13 \) is 3.625 decimal or 11.101 binary.

The above algorithm is easily implemented in a binary machine. Considering only positive numbers, the characteristic can be generated by the following possible method. The machine word is shifted until the most significant "one" appears in the left-most position. A counter counts down from \( n \) (where \( n \) is the number of bits in the machine word) one count for each shift. The end result is that the
bits to the right of the most significant bit contain the mantissa and the counter contains the characteristic.

Consider the example in Figure 15 which illustrates the multiplication of 61 by 67 utilizing the technique presented above. The binary representation of $67 = 0100011$ is stored in the A register and $61 = 00111101$ is stored in the B register. The counter associated with each register is set at the number of bits minus one (i.e. $7 = 111$). At $t = 0$ both registers are shifted simultaneously and the counters decremented. At this point a 1 appears in the most significant bit (MSB) of A so its clock must be disabled for the succeeding shift time states. In $t = 2$ a 1 appears in the MSB of the B register and the process of determining the characteristics and the mantissa is complete. During $t = 3$ the A and B registers can be transferred to other registers in the manner shown or applied directly to the adder inputs as shown. The adder result is illustrated in $t = 4$. Point $t = 5$ consists of decoding the four bit field to the right of the binary point and inserting the new mantissa directly to the right of the decoded result. The actual result desired from the above multiplication algorithm is $1110100000 = 4000$. This results in an error of -2.1%.

Consider a binary number which can be written:
### Figure 15

**Example of Mitchell's Multiplication Method**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>t=0</td>
<td>01000011</td>
<td>00111101</td>
</tr>
<tr>
<td>t=1</td>
<td>111</td>
<td>111</td>
</tr>
<tr>
<td>t=2</td>
<td>10000110</td>
<td>01111010</td>
</tr>
<tr>
<td>t=3</td>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>t=4</td>
<td>1100000110</td>
<td>11110100</td>
</tr>
</tbody>
</table>

Addition Result: 1011111101010

Decoded Result: 00111110100000 = 4000
\[ N = \sum_{i=0}^{\infty} 2^i z_i \quad \text{where } z_i = 0 \text{ or } 1 \quad (31) \]

Since \( z_k \) is the most significant bit we may, without loss in generality, assume that it is a 1 and factor out the \( 2^k \) term

\[ N = 2^k \left( 1 + \sum_{i=0}^{\infty} 2^{i-k} z_i \right) \quad (32) \]

Let the term

\[ 2^{i-k} z_i = x \quad (33) \]

Note that the above term corresponds to the "mantissa" term which falls to the right of the most significant "one" bit.

By substituting (33) into (32) we have:

\[ N = 2^k (1 + x) \quad (34) \]

The binary logarithm of this number is

\[ \log_2 N = k + \log_2 (1 + x) \quad (35) \]

and the approximate binary logarithm is

\[ (\log_2 N) = k + x \quad (36) \]

The logarithm of the product of two numbers is equal to the sum of the logarithms of the multiplicand and the multiplier.

\[ \log_2 P = k_1 + \log_2 (1 + x_1) + k_2 + \log_2 (1 + x_2) \quad (37) \]

Taking the antilogarithm, we have

\[ P = 2^{k_1+k_2} (1 + x_1)(1 + x_2) \quad (38) \]

where the subscripts 1 and 2 refer to the multiplicand and
the multiplier respectively. The approximation for the above logarithm is

\[ \lg P^* = k_1 + x_1 + k_2 + x_2 \]  

(39)

When adding the two approximate logarithms together to form the product, the condition can arise where there will be a carry from the mantissa into the characteristic. As a result of this, equation (39) must be broken up into two parts which account for the \( x_1 + x_2 < 1 \) condition and the \( x_1 + x_2 \geq 1 \) condition. The reason for this is evident if we refer to Figure 12. It can be seen that the number of fractional increments in the mantissa portions of the graph increases as the magnitude of the number increases. As a result, a carry which increases the characteristic should also be accompanied by a corresponding decrease of the mantissa. When \( x_1 + x_2 < 1 \), there is no carry into the characteristic and equation (39) remains valid. When \( x_1 + x_2 \geq 1 \), however, a carry into the characteristic is called for and a 1 should be subtracted from the mantissa. As a result, equation (39) should be rewritten into two parts as follows:

\[ \lg P^* = k_1 + k_2 + (x_1 + x_2) \quad \text{for} \quad x_1 + x_2 < 1 \]  

(40)

\[ \lg P^* = 1 + k_1 + k_2 + (x_1 + x_2 - 1) \quad \text{for} \quad x_1 + x_2 \geq 1 \]  

(41)

Taking the antilogarithm of (40) and (41) respectively, we have:
\[ P' = 2^{k_1+k_2}(1 + x_1 + x_2); \quad \text{for } x_1 + x_2 \leq 1 \quad (42) \]

\[ P' = 2^{k_1+k_2}(x_1 + x_2); \quad \text{for } x_1 + x_2 \geq 1 \quad (43) \]

Taking the true product term in equation (38) and expanding it yields:

\[ P = 2^{k_1+k_2}(1 + x_1 + x_2 + x_1x_2) \quad (44) \]

Since we now have expressions for the true expected product and the actual attained product, we are now in a position to evaluate the maximum error which can result from the use of Mitchell's method. The fractional error can be expressed as:

\[ E = \frac{P' - P}{P} = \frac{P'}{P} - 1 \quad (45) \]

For \( x_1 + x_2 \leq 1 \)

\[ E_1 = \frac{2^{k_1+k_2}(1 + x_1 + x_2) - 1}{2^{k_1+k_2}(1 + x_1)(1 + x_2)} = \frac{(1 + x_1 + x_2)}{(1 + x_1)(1 + x_2)} - 1; \quad \text{for } x_1 + x_2 \leq 1 \quad (46) \]

For \( x_1 + x_2 \geq 1 \)

\[ E_2 = \frac{2^{k_1+k_2}(x_1 + x_2)}{2^{k_1+k_2}(1 + x_1)(1 + x_2)} - 1; \quad \text{for } x_1 + x_2 \geq 1 \]

\[ = \frac{2(x_1 + x_2)}{(1 + x_1)(1 + x_2)} - 1; \quad \text{for } x_1 + x_2 \geq 1 \quad (47) \]

In order to evaluate these two expressions more easily, let \( x_1 + x_2 = s \). Rewriting (46) we have:
\[ E_1 + 1 = \frac{1 + s}{1 + s + x_1 x_2}; \quad \text{for } s = 1 \]
\[ = \frac{1 + s}{1 + s + x_2 (s - x_2)}; \quad \text{for } s = 1 \]  \hspace{1cm} (48)

Taking the derivative of \( E_1 - 1 \) and equating it to 0, in order to find the maximum error, we have:

\[ \frac{d(E_1 - 1)}{dx_2} = \frac{-(1 + s)(s - 2x_2)}{[1 + s + x_2(s - x_2)]^2} = 0 \]

Thus,

\[ s - 2x_2 = 0 \]
\[ s = 2x_2 \]
\[ x_1 + x_2 = 2x_2 \]
\[ x_1 = x_2 \]  \hspace{1cm} (49)

This tells us that the maximum error will occur when \( x_1 \) equals \( x_2 \). Since \( x_1 + x_2 \leq 1 \), \( s \) will have a range from 0 to 1. Evaluating (48) at the lower of these two values results in:

\[ (E_1 + 1) = \frac{1 + 0}{1 + 0 + x_2(0 - x_2)}; \quad s = 0 \]

Since \( x_1 + x_2 = 0 \) and since the only positive values of \( x_1 \) and \( x_2 \) which can satisfy the relation \( s = 0 \) are \( x_1 = x_2 = 0 \), we can finish evaluating \( E_1 + 1 \).

\[ (E_1 + 1) = 1 \]
\[ \text{or} \quad E_1 = 0 \]
Evaluating (48) at \( s = 1 \) results in

\[
(E_1 + 1) = \frac{1 + 1}{1 + 1 + x_2(1 + x_2)}; \quad s = 1
\]

Since \( x_1 + x_2 = 1 \), and since \( x_1 = x_2 \) by equation (49), the only value of \( x_1 \) and \( x_2 \) which satisfies these conditions is \( \frac{1}{2} \). Thus

\[
E_1 + 1 = \frac{1 + 1}{2 + \frac{1}{2} - \frac{1}{2}}; \quad s = 1
\]

\[
= \frac{8}{9}
\]

Thus

\[
E_1 = -\frac{1}{9} = -11.1\%
\]

Going through the same analysis for \( E_2 \), we find that the range in error is the same, i.e. 0 to -11.1%. This indicates that the error incurred by using this technique can be rather high, and while such error might be tolerated in some applications, it cannot be tolerated in a general purpose computer.

The error can be reduced if we recognize the difference between equations (42) and (43) in relation to equation (44). Since equation (44) represents the actual expected product and equations (42) and (43) represent the approximated product, the following equations result if (42) is subtracted from (44), and (43) is subtracted from (44) respectively:
error_1 = 2^{k_1+k_2}(x_1x_2); \text{ for } x_1 + x_2 \leq 1 \quad (50)

error_2 = 2^{k_1+k_2}[1 + x_1x_2 - (x_1 + x_2)]; \quad (51)

for x_1 + x_2 \geq 1.

Equation (50) is a simple correction term, and can be accomplished with two add cycles, one to multiply x_1 and x_2, in the same manner as discussed in the paper, and another to add the result to the original product. Equation (51) cannot be accomplished in the same two steps. However if we note that:

1 + x_1x_2 - x_1 - x_2 = (1 - x_1)(1 - x_2)

and 1 - x_1 = 2's complement of x_1

and 1 - x_2 = 2's complement of x_2

then:

error_2 = 2^{k_1+k_2}(x_1\overline{x_2}) \quad (52)

which can be accomplished in two cycles.
CONCLUSION

This report has presented several multiplication techniques which have, as their unifying feature, the ability to perform the multiplication operation in one cycle. The techniques presented in this report illustrate three methods by which the multiplication process can be speeded up. Briefly stated, these three methods are 1) hardware manipulation, 2) multiplier recoding, and 3) abandonment of 'classical' methods.

The first of the above methods is evident in the presentations on the Prioste, Carry-Save, Dadda, and Richards multipliers. All of these methods seek to implement the multiplication problem in terms of, what the author calls, the 'classical' approach. This is the sum of the shifted partial products. The difference between the above methods is not an algorithmic difference but a difference resulting from the choice and interconnection of the hardware components. The Prioste and Richards multiplier are alike in their use of full word adders but the approaches differ as to their interconnection. The Carry-Save and the Dadda multipliers differ from the Prioste and Richards multipliers in their use of full and half adder circuits and they differ from each other in the way that these circuits are interconnected. By
the manipulation of hardware, each of the above approaches strives to achieve greater speed than the others.

The second method utilized for increasing the speed of simultaneous multiplication is evident in the presentation of the Wallace multiplier. Although the Wallace multiplier can be considered as a classical multiplier, his use of multiplier recoding (pre-encoding) succeeds in reducing the number of operands and thus increasing the speed of the total multiplication process.

The third method for increasing the speed of simultaneous multiplication was demonstrated by the approaches of Ling, Chen and Mitchell. These methods abandon the restrictions of the classical method and seek speed increases in more novel methods of multiplication. These approaches are relatively recent and, in the author's estimation, represent significant improvement in the "state of the art".

This report has demonstrated certain areas which are open for further research. The first of these is motivated by the fact that, to the author's knowledge, there is no mention of a 'logical' multiplication anywhere in literature. Assuming no limitations on hardware fan-in and fan-out, it is theoretically possible to implement the multiplication process with only two levels of gating if it is implemented in a sum of products or product of sums configuration. This is
certainly possible for small numbers. However as the size of the numbers increases, the two level realization becomes impractical due to the fan-in and fan-out problems. The task of reducing the multiplication truth table into a set of realizable boolean equations becomes quite unmanageable as the word length increases. However the advent of more efficient computer programs for the generation, decomposition, and minimization of boolean functions may make this problem more manageable. The advantages of this method are not obvious without further analysis but the author feels that the results may give rise to an extremely fast multiplier.

The literature on iterative multiplication abounds in the methods for multiplier recoding. The author's investigation reveals only one simultaneous multiplier which uses the principles of multiplier recoding. This is the Wallace multiplier. Great pains were taken to compare the Wallace and Richards multiplier in order to demonstrate that the advantages of the Wallace multiplier were incurred through his use of recoding. Applying the same recoding techniques of Wallace to the Richards multiplier would result in a multiplier which is faster than the Wallace multiplier. This is an example of only one recoding method applied to two simultaneous multipliers which will result in an increase in speed. The author feels safe in stating that more signifi-
cant reduction in operands can be achieved if the recoding techniques, described in iterative multiplication literature, were applied to non-iterative multiplication.
LIST OF REFERENCES


BIBLIOGRAPHY I


