Abstract - In a modern day CPU, the multiplier has become an important part of an ALU in terms of both power and performance. Modern day multiplier designs provide some enhancements in power and performance over traditional multiplier designs. However, this enhancement, especially from a power perspective, may prove to be inadequate for a low-power application. Using lookup-based tables for multiplication is an attractive alternative to other modern day multiplier applications in order to greatly reduce the amount of power required. Because so much research has been done on reducing the power usage of Read Only Memories (ROMs), lookup-based tables for multiplication can be implemented with a much lower power than traditional designs. While using lookup tables for multiplication has been researched, most approaches still require tables too large for realistic implementation. Both Vinnakota[1] and Ling[4] introduce approaches to reduce the size of their multiplication tables, but neither approach can be used with modern, 32-bit, multipliers feasibly. To achieve multiplication using smaller tables, we introduce a hybrid approach that also utilizes the Karatsuba algorithm to reduce the table size needed for an n-bit multiplication down to a table of n/2-bit addresses and n-bit words. The paper is organized as follows. In section II, we look at past multiplier approaches, including Array Multipliers, Wallace Tree Multipliers, Multipliers with Booth representation, and its significance from a power-limited perspective. In section III, we cover modern ROM design, and estimate latency and power draw for our multiplier's target table size. Section IV briefly covers past table-based multipliers, including Vinnakota's approach using a table of squares. Then in section V, we introduce our hybrid approach and show a possible implementation. We go over the target users and applications for our multiplier in section VI. In Section VII we outline our test procedure and benchmark setup. Section VIII contains performance measurements of our multiplier under various benchmarks. Concluding remarks about our results, as well as speculation into further research is made in section IX.

II. Modern Multiplier Designs

II.a. Array Multipliers

The most direct approach to multiplication implementation is the Array Multiplier. Array Multipliers follow the basic order of multiplication by hand, including partial product generation and partial product summation. Partial products are generated for the multiplicand using the logical AND operator. Every bit of the multiplicand is multiplied with each bit in the multiplier starting from the least
significant bit, and shifted one place for each order of magnitude for the multiplier. The partial products are then summed together in the same way as in hand calculation. The following shows the logic gate for partial product generation and the math for a 4bit by 4bit multiply in hand calculations:

\[
\begin{array}{cccccccc}
& a_3 & a_2 & a_1 & a_0 & b_3 & b_2 & b_1 & b_0 \\
A_1 & & & & & \cdot B_1 & & & \\
& a_3 b_3 & a_2 b_3 & a_1 b_3 & a_0 b_3 & a_3 b_2 & a_2 b_2 & a_1 b_2 & a_0 b_2 \\
& a_3 b_1 & a_2 b_1 & a_1 b_1 & a_0 b_1 & a_3 b_0 & a_2 b_0 & a_1 b_0 & a_0 b_0 \\
& a_3 b_0 & a_2 b_0 & a_1 b_0 & a_0 b_0 & & & & \\
\end{array}
\]

Figure 1: Array Multiplication

An important issue to note for the Array multiplier is that there is a Carry signal that is propagated through the array of full adders starting from the 2\textsuperscript{nd} least significant bit of the product, \( p_1 \). As a result, the critical path of this multiplier design lies in the carry chain that goes from the 2\textsuperscript{nd} least significant partial product down to the most significant partial product. For an N-bit by N-bit multiplication, this is approximately \( 2N-1 \) Full-adders that must be passed through before the final output is ready. Another issue with the Array Multiplier architecture, is that it requires \( N^2 \) Full adders and \( N^2 \) AND gates. Conventional Full-Adder designs in standard CMOS take 24 transistors, and AND gates take 4 transistors. This comes out to \( 28^*N^2 \) transistors in order to implement an N-bit by N-bit Array multiplier. The switching power needed for an Array multiplier is also relatively high, as there are no optimizations in terms of partial product reduction or critical path. The only benefits of the Array multiplier are simplicity for generation and understanding. The paper by Muhammad et. Al goes into some of the characteristics of Array Multipliers in more detail. [5]

II.b. Wallace Multipliers

The Wallace Multiplier is an improvement off of the Array Multiplier in order to reduce the critical path of the multiplier [2]. It is based on the premise of trying to combining the partial products as quickly as possible, instead of waiting for less significant bits to generate a carry signal. A Wallace multiplier consists of two parts: a Wallace Tree for reducing and combining the partial products, and a Final Adder to generate the actual product. In a Wallace multiplier, the number of partial products generated is the same as in the Array Multiplier. Thus there are still \( N^2 \) AND gates required for an N-bit by N-bit multiplication. The difference lies in the way the partial products are added together in the Wallace tree.

In the first layer of a Wallace tree, partial products of the same bit-weight are grouped together. This is the same as grouping each column in the hand calculations of the partial products, thus the 4-bit by 4-bit multiplication in the previous figure would have 8 groups. These groups are then added together using as many full adders as necessary. The next levels in the Wallace tree reduce the resulting sums and carries using additional full-adders. This is continued until there are at most 2 bits of each weight. This result is then passed to the second stage of the Wallace Multiplier,

\[\text{Figure 2: Wallace Tree}\]
Wallace tree for one branch of the Wallace tree. These levels are referred to as reductions, for each level takes 3 inputs (A input, B input, Carry-in), and reduces it to 2 outputs (Sum and Carry-out).

Due to the fact that the Carry signal in a Wallace tree propagates down each level, the critical path in a Wallace tree is significantly reduced. In an Array Multiplication scheme, the Carry signal must propagate across all N bits of both the multiplicand and multiplier. In a Wallace scheme, because the partial products are grouped by bit-weight, much of the initial addition occurs in parallel. The number of levels required depends on the bit-weight group with the largest number of partial products and is logarithmically related to the number of bits being multiplied. The delay through the Wallace tree is equal to the delay of a full adder times the number of levels. The final Carry-Propagate adder is approximately 0.5*N bits in size due to the reduction that occurred in the Wallace tree. Thus the delay through a Wallace multiplier is roughly approximated by 0.5*N+log(N).

Due to the less uniform arrangement of adders however, the Wallace multiplier uses a slightly greater number of Full Adders to perform its task. This is because of the fact that the grouping of partial products by their bit weights often ends up not completely utilizing a Full-Adder. Often, a Half-Adder is used for these cases, which occurs in about 1/3 of the bit weights. Despite this, the switching power for a Wallace Multiplier is approximately the same as that of an Array Multiplier.

**II.c. Modified Booth Encoding**

Booth encoding for multipliers is an idea that allows for the reduction in the number of partial products and adding stages depending on the input vector, by taking in multiple bits at a time [16]. In a Radix-2 Booth encoder, chains of 1’s are converted to a 1 of the next higher bit-weight and a -1 at the lowest bit-weight. For example, the 4-bit number for 7 is: 0 1 1 1. In booth encoding, this would look like 1 0 0 -1. Doing the math for the new number we see that it represents 8 - 1, which is also equal to 7. The number of non-zero partial products decreases to 2 instead of 3. The reduced set of partial products can then be used with any type of multiplier. This is useful because the speed and power are both related to the total number of partial products. A major disadvantage of the Booth encoding, however, is that the reduction of the partial products depends completely on the input vector.

A solution to this is the modified booth encoding, which reduces the number of partial products by half in all cases. It does this by examining 3 bits, but skipping 2 bits at a time. In a Radix-2 modified Booth encoding scheme, 2 bits of the multiplier are converted into a 0, ±1, or ±2 by examining the 2 bits and the one bit preceding it. The ±2 versions of the multiplicand are done by a simple shifting of bits. As a result of the encoding, the multiplicand is converted through shifting or inversion into those forms and a MUX or some other form of selector is used to select according to the multiplier bit pattern. The following truth table, Figure 3, determines the conversion based on the digits of multiplier Y (where i goes from 0 to N-1 in even multiples) and adding the resulting partial product of multiplicand X. Note that there is an implicit Y_{i-1}=0 for the first group of digits. The signals Neg, Two, and Zero are used for the MUX selects to choose the correct bit of the multiplicand. Numbers in parenthesis are don’t cares:

<table>
<thead>
<tr>
<th>Y_{i+1}</th>
<th>Y_i</th>
<th>Y_{i-1}</th>
<th>Factor</th>
<th>Neg</th>
<th>Two</th>
<th>Zero</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(0)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>+X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>+X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>+2X</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-2X</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-X</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-X</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>(1)</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 3: Modified Booth Encoding

For example, the number 6 in binary is: 0 1 1 0
In modified Booth encoding it becomes: 2 -2
This means that there is a 2 in the 4’s digit and a -2 in the 1’s digit, which gives us 8 - 2 = 6, which verifies that it is identical.

It is important to note the trade-offs associated with using a modified Booth
encoding. The biggest advantage of the modified Booth encoding is definitely the reduction of partial products by half, which can have a significant impact especially for large multipliers as scaling continues. For an Array multiplier with modified Booth encoding, the delay time is halved and switching and leakage power is reduced to 25% of the non-Booth encoded version for the adder array. For a modified Booth encoded version of a Wallace multiplier, performance is improved by a little less than 50%, but this proves to be quite a lot compared to a regular Array multiplier. For the switching power, a modified Booth version of the Wallace multiplier, there is approximately a 30% reduction in the Wallace tree and final adder.

However, these gains are offset by some of the overhead required for Booth encoding. First of all, the numbers 0, ±1, or ±2 of the multiplicand need to be generated. Second, they need to be selected, through a MUX or other logic gates depending on which input sequence is seen. On the bright side, this computes to a constant overhead, as each of these requirements can be done in parallel. Thus for large bit-widths, and as processor scaling grows, the benefits outweigh the disadvantages by far. In addition, much research has gone into improving this “encoding” requirement for the Booth algorithm, including Redundant Binary encoding [14], Predetermined coefficients [15], and MUX select Booth encoders [7] [9].

II.d. Modern Multipliers from a Low Power Perspective

Wallace Trees are able to improve the performance off of an Array multiplier, and with the addition of modified Booth encoding power and performance are able to be improved even further. However, this is not sufficient for some low-power applications, which are becoming more and more prevalent. At the 0.25µm Technology node, a 32-bit by 32-bit Wallace multiplier with modified Booth encoding had a delay of approximately 4ns, and consumed approximately 1000 pJ. Our study of the ROM based multipliers will compare to this Wallace multiplier for analysis.

III. Modern Read Only Memory design

There has been significant research into ROMs with the intent of reducing power, especially for mobile systems like laptops or PDAs[10][11] or wireless communication[17]. ROM density has also been looked into and ROM sizes will likely decrease [18][19]. Performance in a ROM is largely dependent on how high the voltage is, so a low-power ROM can also exhibit high performance with higher voltages (this cancels much of the power benefit, however). From various sources, we tried to estimate the latency of a 16-bit address (64K), 32-bit word (2 Mb) sized table which we use in our proposed implementation.

We chose to normalize our simulation to a .25µm process size. With ROMs at this process size we see a 6ns delay (for 1K x 32-bit), consuming 3.53 mW of power for a Charge-Recycling ROM [10]; and a 4.2ns delay (for 4K x 32-bit), consuming 8.2mW for a ROM with a Charge-Sharing Capacitor [11]. The same Charge-Sharing Capacitor ROM consumed 3.24 mW when clocked slower with a 10ns delay. From this, we estimated that one could construct a 64K x 32-bit table with a 7ns delay, and consuming 20mW while active.

IV. Past Table-based Multipliers

There has been little research into the area of table-based multipliers. Ling [4] and Vinnakota [1] both introduce theoretical designs using the following general property:

Let \( x = \text{floor}((A+B)/2) \)
Let \( y = \text{floor}((A-B)/2) \)

if A and B are both odd or both even
\[ A*B = x^2 - y^2 \]
if only one of A or B is even
\[ A*B = x^2 - y^2 + B \]

This property is very easily implemented on a computer, floor(N/2) simply equals N shifted right by 1. The least significant bit of A and B can be put through just one XOR gate to determine whether or not to add B to the final product. Using this reduces an n-bit multiply from needing a \( 2^n * 2^n \) by 2n sized table to a \( 2^n \)
by 2n sized table, requiring two lookups and four additions. x and y can be computed in parallel, as can $x^2$ and $y^2$ (although this requires two tables).

Vinnakota further refines this approach with a split-table idea. From the paper:

“Let $N_1$ and $N_2$ be two binary numbers such that the $k$ least significant bits of $N_1$ and $N_2$ are identical. Then, the least significant ($k + 1$) bits in $N_1^2$ and $N_2^2$ will also be identical. …

One possible two-table partition is:
1) Table A with $2^{6}(= 64)$ 7-bit entries. The most significant seven bits of the squared values of the numbers in the range 0 to 63 are stored in this table.
2) Table B with $2^{4}(= 16)$ 5-bit entries. The least significant five bits of the squared values of all the numbers in the range 0 to 15 are stored in this table”

This approach uses 528 bits instead of 768 bits. The somewhat reduced size of the largest table leads to faster overall lookups as well.

With our target table size of $2^{16}$ by 32-bit, a possible split-table approach could be a $2^{16}$ by 8-bit table: $2^{19}$ bits; a $2^{12}$ by 8-bit table: $2^{15}$ bits; and a $2^8$ by 16-bit table: $2^{11}$ bits. This gives a total of 559,104 bits compared to 2,097,152 bits—nearly a factor of 4 in bit savings.

Research has also been done using multiplication with Karatsuba’s algorithm [8][22][20]. Briefly, the algorithm states:

Let A be an n-bit number and equal $\{A_1, A_0\}$ where $A_1$ represents the top $n/2$ bits and $A_0$ represents the bottom $n/2$ bits of A.

Let B similarly equal $\{B_1, B_0\}$

$A * B$ then equals

\[
(A_1 * B_1 << n) + (A_1 * B_0 << n/2) + (A_0 * B_1 << n/2) + (A_0 * B_0)
\]

$N << X$ denotes a left shift of $N$ by $X$ bits.

In all, using Karatsuba’s algorithm reduces an N-bit multiply into 4 N/2-bit multiplies and 3 additions.

Section V: Our Implementation

While Vinnakota and Ling significantly reduced the size of table need to allow table-based multiplication, modern processors require at least 32-bit multiplication. With either the table of squares approach of Karatsuba’s algorithm, the number of entries in the table will be $2^{32}$. A thought was to use a hybrid approach to further reduce the table size to a manageable 64K x 32-bit lookup (which can be further reduced by splitting the table). With this hybrid approach, a product $C$ from operands $A = \{A_1, A_0\}$, $B = \{B_1, B_0\}$ can be calculated by finding:

1) Find $X_{ij}$, $X_{10}$, $X_{01}$, $X_{00}$, $Y_{11}$, $Y_{10}$, $Y_{01}$, $Y_{00}$ where

\[
X_{ij} = \text{floor}((A_i + B_j)/2)
\]

and

\[
Y_{ij} = \text{floor}((A_i - B_j)/2).
\]

This uses eight 16-bit addition operations, which can be performed concurrently.

2) Use table lookups to find $X_{ij}^2$ and $Y_{ij}^2$

This uses eight 64K x 32-bit table lookups, which can be performed concurrently.

3) Find $C_{11}$, $C_{10}$, $C_{01}$, $C_{00}$, where

\[
C_{ij} = x_{ij}^2 - y_{ij}^2 + (B_j \&\& (A[0] \text{ XOR } B[0]),
\]

where && is a masking operation, and $N[0]$ is the least significant bit of $N$.

This uses eight 32-bit additions, where each pair of two can be performed concurrently with another pair.

4) Calculate $C = (C_{11} << 32) + (C_{10} << 16) + (C_{01} << 16) + C_{00}$

This uses three 32-bit additions, which must be performed sequentially.
We note that this design can be easily pipelined, giving us the following rough model:

![Diagram of pipelined implementation](image)

The dark bold lines denote possible pipeline stage cutoffs. Each $C_{ij}$ is given a phase of the pipeline, so $X_{ij}$ and $Y_{ij}$ are calculated in parallel along with their squares (two cycles), then $C_{ij}$ itself. As each $C_{ij}$ leaves this pipeline it can be sent to an accumulator to calculate $C$. With this we estimated roughly our implementation would take a total of 24 ns, or 6 times as long as a conventional multiplier.

**Section VI: Testing Goals and Procedure**

Our goal from testing was to find out whether the slower speed would justify using this multiplier over a traditional one for mobile users. Laptop users especially are generally interested in multimedia or office applications, so we wanted to weigh the results of those applications or similar benchmarks more heavily.

Tests were conducted using Simplescalar ([www.simplescalar.com](http://www.simplescalar.com)). Modifications were made to sim-outorder.c, where we varied the number of cycles required to perform integer and floating point multiplication. The floating point multiplier was given a one cycle delay, since the original code had it, presumably for rounding calculations. The general, one-cycle multiplier thus had a 1-cycle latency for integer multiplication and 2 for floating. Our proposed 6-cycle multiplier had a 6 cycle latency for integer multiplication and 7 for floating point multiplication.

The machine used was an x86 processor running a Gentoo distribution of Linux through Windows using CoLinux ([www.colinux.org](http://www.colinux.org)). The benchmark used was SPEC 2000, and little endian binaries (required for linux) were obtained from the MIRV project [21], compiled using gcc –O2 optimization. The following tests were run, separated into those we thought were important for our target application types:

- **bzip**: a compression benchmark—compression is important because many media files are compressed and require on-the-fly decompression.
- **gzip**: another compression benchmark.
- **ijpeg**: image compression/decompression—many laptop users expect to be able to watch movies or view pictures.
- **mesa**: 3-D graphics library—we felt this was a representative benchmark of video rendering.
- **parser**: word processing—most mobile devices (laptops, PDAs, cell phones…) implement some sort of word processing or language parser.

The tests that we ran but did not consider significant were:

- **art**: object recognition in a thermal image.
- **gcc**: GNU C compiler.
- **go**: go playing program.
- **li**: lisp interpreter.
- **m88ksim**: chip simulator.
- **mcf**: vehicle scheduling.
- **vortex**: object oriented database.
- **vpr**: placement and routing for Field-Programmable Gate Arrays.
*Note: some benchmarks froze up when we ran them for non-trivial (>5000 instructions) lengths, their results are omitted. None of the omitted benchmarks had significant differences (due to the short run length), nor were they deemed as being within the target application group.

**Section VII: Results**

Figure 4 shows results of using a regular 1-cycle multiplier compared to our proposed 6-cycle multiplier. Figure 5 shows the same data in a bar graph.

<table>
<thead>
<tr>
<th>Bench</th>
<th>1-cycle Count</th>
<th>6-cycle Count</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Art</td>
<td>13773</td>
<td>13773</td>
<td>0.0%</td>
</tr>
<tr>
<td>Bzip</td>
<td>1056164</td>
<td>1056164</td>
<td>0.0%</td>
</tr>
<tr>
<td>Gcc</td>
<td>236404</td>
<td>237056</td>
<td>-0.29%</td>
</tr>
<tr>
<td>Go</td>
<td>157684</td>
<td>166180</td>
<td>-5.39%</td>
</tr>
<tr>
<td>Gzip</td>
<td>1057523</td>
<td>1057523</td>
<td>0.0%</td>
</tr>
<tr>
<td>Ijpeg</td>
<td>21404</td>
<td>21414</td>
<td>-0.05%</td>
</tr>
<tr>
<td>Li</td>
<td>171416</td>
<td>171691</td>
<td>-0.16%</td>
</tr>
<tr>
<td>M88ksim</td>
<td>60163</td>
<td>61128</td>
<td>-1.63%</td>
</tr>
<tr>
<td>Mcf</td>
<td>9031</td>
<td>9031</td>
<td>0.0%</td>
</tr>
<tr>
<td>Mesa</td>
<td>489319</td>
<td>492315</td>
<td>-0.61%</td>
</tr>
<tr>
<td>Parser</td>
<td>34146</td>
<td>34150</td>
<td>-0.01%</td>
</tr>
<tr>
<td>Vortex</td>
<td>47996</td>
<td>48161</td>
<td>-0.34%</td>
</tr>
<tr>
<td>Vpr</td>
<td>34618</td>
<td>34618</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

Notes: gcc and go were run at 300,000 maximum instructions, since the simulation would hang if it was not limited.

As is shown, the performance differences are largely negligible. The benchmarks which exhibited a greater than 1% slowdown were the go-playing program and the circuit simulator—neither of which is likely to be of much importance to most laptop or PDA users.

For determining power, we used the 4 ns cycle time and number of cycles to find the length of the program; then summed the total wattage of all the elements of the multiplier to gauge the power savings within the multiplication unit.

For the combinatorial multiplier, power consumption for a .25um process we estimated to be .25W.

For our proposed multiplier, each of the two tables consumed about 20mW, while each 32-bit adder would consume 15mW (2/32 the power of the combinatorial multiplier). With 2 16-bit adders at the front and two 32-bit adders at the end, we get a total power consumption of 65mW. Figure 6 shows the total program time and total energy consumption over all benchmarks, Figure 7 shows total program time and total energy consumption over the benchmarks which were indicated as more
relevant. It is important to note that these figures are for the multiplier only, and do not include the rest of the system.

In all, we see little difference in narrowing our benchmark suite—this is probably because many irrelevant benchmarks contributed cycles which hide the impact of the more relevant benchmarks.

---

### Program Times and Energy Consumption for all Benchmarks

**Normalized to Base Multiplier**

![Figure 6](image)

### Program Times and Energy Consumption for Target Benchmarks

**Normalized to Base Multiplier**

![Figure 7](image)
VIII. Conclusion

We have shown that, by combining the use of a table of squares with the Karatsuba algorithm it is feasible implement lookup-table based multiplication. Furthermore, out of the benchmarks examined, the performance penalty taken by the multiplier had little impact overall—however, further research with more specific benchmarks is highly desirable. The estimated power consumption of the table based multiplier is far less than that of a general multiplier; and, with more or less equivalent program speeds, we see it is possible to achieve a significant reduction in the energy drawn by the multiplier.

There is much further work that can be done with this approach to multipliers. Optimizations can be made to the design, such as re-organizing the pipeline to make for more even pipeline stages. Using Vinnakota’s split-table approach could lead to a significant savings in ROM area and latency. Finally, more measurements need to be made. An important measurement to be made would be finding out the impact of this power change on the system as a whole, instead of just the multiplier. Finding out the actual power draw on real silicon would be another avenue to explore.

IX. References


