Optimized Memristor-Based Multipliers

Lauren Guckert, *Student Member, IEEE*, and Earl E. Swartzlander, Jr., *Life Fellow, IEEE*

***Abstract*— Since memristors came to the forefront of research, minimal work has explored their application to computer arithmetic. This paper proposes two memristor-based implementations of an N-bit shift-and-add multiplier, one using IMPLY operations and a second using MAD operations. The optimized IMPLY-based implementation reduces the baseline delay from $2N^2 + 29N$ steps and 17N+3 memristors to $2N^2 + 21N$ steps and 7N+1 memristors. A second implementation is proposed that is constructed from MAD gates, a lower-area, lower-delay alternative to IMPLY logic. This design performs an N-bit multiplication in $N^2 + N$ steps with 5N memristors and 3N+2 drivers. Both designs require fewer steps and less than 1/6 of the number of components of a traditional CMOS design. Finally, both of the implementations are extended to implement radix-2 Booth multipliers. The IMPLY design only increases by 1 step per iteration and 2N memristors and drivers. The MAD design increases by N memristors and 6N switches but maintains the same delay as the shift-and-add multiplier. Both designs maintain a lower area and lower delay than the CMOS equivalent.**

***Index Terms*— Booth's algorithm, driver circuitry, IMPLY logic, MAD gates, memristors, multipliers, shift-and-add multipliers.**

I. Introduction

SINCE memristors were first presented by Leon Chua [1], research has explored the use of these devices in the context of modern system design. The most prominent benefit of memristors is the ability to hold data with low area and high density. Thus, the majority of research in memristors has focused on their application to memory [2]–[6]. However, recently research has shown that memristors can be used to implement all Boolean operations using the IMPLY operation [7]. Since this finding, an orthogonal path of research has begun to explore arithmetic applications for memristors [8]–[23]. Some of these works have focused on the IMPLY operation [8]–[13], while others propose alternative approaches that offer lower area or delay [14]–[23]. However, all of the preliminary work that has been done has focused on individual logic gates and small circuits such as adders.

Although these initial works are useful as foundational pieces, they are not directly applicable to modern systems. Modern systems often require complex arithmetic units such as large N by N multipliers to perform fast multiplication. In this work, a novel implementation of an N-bit shift-and-add multiplier is presented that is built upon the IMPLY operation. The design is optimized to overlap Boolean operations between the arithmetic components and simplify the multiplexer logic. The N by N multiplier design requires $2N^2 + 21N$ steps and 11N+6 memristors with 7N+6 drivers. This is fewer steps than required in the CMOS design, which is $2N^2 + 23N$. Then, further optimizations are performed to leverage characteristics specific to the IMPLY operation. The traditional layout of a shift-and-add multiplier can be simplified to remove the shift registers entirely, reducing the design's area to almost that of an IMPLY ripple carry adder - 7N+1 memristors, 7N drivers, 8N-1 switches, and $N + 1$ resistors. This is over an 80% reduction in the number of components in the traditional CMOS shift-and-add design. It is also shown that this design has the benefit over CMOS of pipelining additions. A new addition can begin every 24 steps in the design, making the effective latency of a multiplication as low as 24N steps, O(N).

Although the design is aggressively optimized, Boolean operations in IMPLY gates still suffer from the need to be serialized. For a long series of gates, this leads to high critical path delays. As the complexity of the system increases, these delays are magnified. As an alternative, this work leverages a new approach to memristor gate design, MAD gates, that overcomes these issues and provides lower area and delay. An N-bit shift-and-add multiplier constructed from MAD gates is presented and optimized to require fewer components and steps than the IMPLY approach. The total complexity is 5N memristors and 3N+2 drivers for an N by N multiplier and the latency is N^2+N steps. This is less than half the latency of the IMPLY and CMOS designs while requiring significantly fewer components, too. MAD gates also allow for the multiplier to pipeline additions, beginning a new addition every 4 cycles. Thus, at high utilization, the presented MAD multiplier can complete a multiplication every 4N steps, less than 1/6 the time of the pipelined IMPLY design.

Both of the shift-and-add designs are extended to implement Booth's algorithm in radix-2. For the IMPLY-based approach, this requires an additional 2N memristors and 1 step per addition for the pipelined model. For the MAD-based approach, the delay remains the same and the complexity increases by only N memristors, 1 driver, and 6N switches. Both designs continue to use fewer components than the CMOS counterpart.

II. Prior Work

The original popularized approach to memristor logic employs the IMPLY operation, which can be successfully performed with memristors using two memristors and a load resistor [2]. Prior work showed how the IMPLY operation can be extended to perform all Boolean operations by performing a series of IMPLY steps on a series of memristors [7]. The Boolean operations can in turn be concatenated

Manuscript received May 10, 2016; revised August 3, 2016 and August 30, 2016; accepted September 3, 2016. Date of publication January 9, 2017; date of current version January 26, 2017. This paper was recommended by Associate Editor G. Masera.

The authors are with the Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX 78712 USA (e-mail: lguckert@utexas.edu; e.swartzlander@ieee.org).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSI.2016.2606433

to achieve more complex circuits [8], [9]. Many subsequent works have focused on implementing arithmetic structures using the IMPLY operations [8]–[12]. However, these works do not perform optimizations or leverage all the opportunities for parallelism in the designs. For example, some prior works [8]–[10] propose IMPLY adder designs but they are theoretical and not implemented or optimized. Other work has focused on improving the Boolean operations only [11], without any application. Still more works have proposed optimized shift registers [12] and ripple carry adders [13] constructed from IMPLY operations but limited the scope to these designs. Throughout all of these works, almost none have explored the implementation of more complex arithmetic units, such as multipliers. The only exception is in [8], where a IMPLY-based design of an array multiplier is presented. However, even in this work, the design is not optimized and is only analyzed at a mathematical level.

Additionally, some of these IMPLY based implementations have been criticized for their multi-step operations and interference [24]–[26]. This is a consequence inherent to the IMPLY operation due to the high area and delay for some Boolean operations. As designs grow in complexity, these issues become magnified. This has led to research in alternative approaches to memristor-based Boolean logic, including hybrid approaches [14], [15], MAGIC gates [5], CRS cells [6], and other novel approaches [2]–[4], [16]–[22]. However, each of these approaches has issues in terms of their completeness, applicability, or latency and area. Some of the works are limited to logic-in-memory in the crossbar context and are logically incomplete [2]–[4]. Others are prohibitive in terms of their reliability and ability to be fabricated [18]–[22].

To overcome these limitations, MAD gates [23] have been proposed. Similar to IMPLY, MAD gates use a notion of applying select voltage signals on the terminals of adjacent memristors. However, they leverage voltage division and threshold detection to achieve low latency, low area operations. MAD gates are capable of performing all Boolean operations in only a single cycle and with a standardized cell containing 3 memristors. They can also perform the COPY operation in a single cycle and 2 memristors. Lastly, because they rely on voltage drivers for their operation, concatenation and high fanout are not issues as they are for other approaches. MAD gates have been used in the arithmetic context to construct optimized ripple carry adders and carry select adders. No prior work has yet explored the application of MAD gates to multipliers.

Many memristor models exist depending on the particular application or type of memristor being targeted [27]–[29]. For all simulations, this work uses the Cadence Virtuoso Tool and the TEaM memristor model [27]. The parameters for the model were selected to match those in the original TEaM work.

III. Design Background

Before introducing the proposed multiplier designs, a background on memristors and the traditional design for a shift-and-add multiplier and a radix-2 Booth multiplier are given. An overview of the IMPLY and MAD methodologies is also given.

A. Memristor Background

Memristors were first introduced by Leon Chua in 1971 [1] and the majority of subsequent research focused on understanding and designing these devices [30]–[32]. Although this is an ongoing effort and the physics of the devices are extremely complex, a brief introduction into memristors is provided.

The term ‘memristor’ comes from ‘memory’ + ‘resistor’ because it is a device with an internal resistance that can change and be remembered. The internal resistance is controlled by the current that flows through the memristor. If the magnitude of the current through a memristor is less than a threshold I_t, the internal resistance of the memristor will remain unchanged. This is useful when a ‘read’ operation is desired—a low-magnitude current can be driven through the memristor and the voltage drop is measured to determine the resistance or ‘memory value’ of the memristor. If the magnitude of the current is greater than I_t, the internal resistance of the memristor will change. If the current is positive, the resistance will decrease, else it will increase. This can be used for a ‘write’ function, changing the internal resistance of the memristor to represent the desired stored value.

Besides the unique behavioral properties of the memristor, they also offer a smaller form-factor than CMOS and fast switching times. Memristors are on the order of 3 nm^2 and can switch in as few as 120 ps, leading to an overall low power consumption. The exact switching time of the memristor depends on multiple factors including the voltage, where a higher voltage leads to a faster switching time.

One of the many difficiulties to understanding memristors is classifying and modeling their physical properties and behavior. Memristors have many complex characteristics, including threshold parameters and doping width. To add to the complexity, memristors have high process variability. The exact response curve for a memristor varies over time and from device to device. It is affected by process technology, past behavior and resistance, interference, and more. This makes it difficult to correctly model memristors or implement designs and calculate circuit parameters such as voltages and currents to provide correct functionality.

Fortunately, there has been continued progress in this area and models have been created to accurately represent the variability of memristors [30]–[32]. Also, HP Labs successfully fabricated memristors and showed that all IMPLY operations can be implemented with them [7]. Research such as this has elevated memristors as practicable, nanoscale structures that can be fabricated for future system designs. Thus, this work is based on the assumption that memristors will become a viable candidate for architecture design in the future and that the current models are accurate to represent their behavior.

B. General Shift-and-Add Multiplier Design

This section provides a general overview of the shift-and-add multiplier to provide context into the specifics of

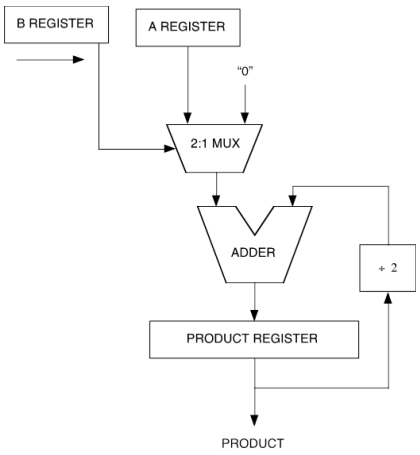

Fig. 1. Schematic for a shift-and-add multiplier.

the optimizations performed for the proposed designs. The multiplier is favored for its low complexity but this comes at the cost of higher latency. A visual representation of this can be found in Fig. 1.

The multiplier performs N iterations, doing a select, addition, and shift in each iteration. Bit 0 in the multiplier register B is sensed in each iteration to determine if 0 or the multiplicand operand A should be added to the running product register. If the 0th bit of the multiplier is 0, A'= 0, else A'= A. After the addition, the value is stored back into the product register. Then both the multiplier and product registers are shifted once to the right. The process repeats for all N bits of the multiplier and the final result lies in the product register.

One common optimization is to combine the multiplier and product registers. Since the product register is resolved one bit at a time and the multiplier is discarded one bit at a time, the multiplier can begin in the rightmost N bits of the product register. This eliminates the N-bit multiplier shift register.

Booth multipliers offer a faster alternative to shift-and-add multipliers with the addition of minimal complexity. A Booth multiplier examines multiple bits of the multiplier at each iteration rather than just one. This work considers the radix-2 Booth algorithm, which examines two bits on each iteration with one bit of overlap between consecutive iterations. Depending on the value of these two bits, a different value is added to the product register. If the two bits match, 0 is added, if the two bits are '01', the multiplicand is added, and if the two bits are '10', the 2's complement of the multiplicand is added. Thus, two of the inputs remain the same as the shift-and-add multiplier, but the 2's complement of the multiplicand is added as a potential operand.

C. IMPLY Methodology

The IMPLY operation has become popularized as an approach to Boolean logic using memristors. The IMPLY operation is an operation which takes two input operands, p and q, and stores the result in the q operand. The operation is designated by '->' or by 'IMP', as in p-> q or p IMP q, and the truth table for the operation is shown in Table I.

TABLE I
TRUTH TABLE FOR THE IMPLY OPERATION

p	q	p ->q
0	0	1
0	1	1
1	0	0
1	1	1

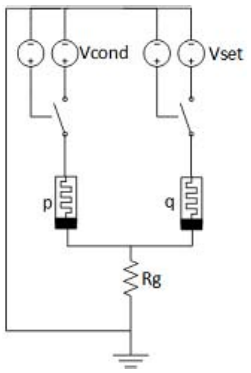

Fig. 2. Circuit for an IMPLY operation.

TABLE II
IMPLY IMPLEMENTATIONS OF BOOLEAN OPERATIONS

Operation	Implementation	Latency	Area
p NAND q	p IMP (q IMP 0)	2	3
p AND q	(p IMP (q IMP 0)) IMP 0	3	4
p NOR q	((p IMP 0) IMP q) IMP 0	5	6
p OR q	(p IMP 0) IMP q	4	5
p XOR q	(p IMP q) IMP ((q IMP p) IMP 0)	8	7
NOT p	p IMP 0	1	2

Memristors can perform the IMPLY operation with two memristors and a load resistor. At a common node, the two memristor operands, p and q, are connected to each other at their p-terminals and the load resistor, Rg, is connected to ground. This can be seen in Fig. 2.

The p and q memristors both serve as inputs to the IMPLY operation. To perform the IMPLY operation, a voltage V_{cond} is applied to the n-terminal of the p memristor while a voltage V_{set} is applied to the n-terminal of the q memristor. V_{cond} and V_{set} are selected such that $V_{cond} < V_{set}$. Also, V_{cond} must be less than the threshold voltage of the memristors and V_{set} must be greater than the threhsold voltage to ensure the value of the result memristor is correctly set. The load resistor, R_g is also included for this purpose. In this work $V_{cond} = 1.6V$, $V_{set} = 2.5V$, and $R_g = 2K$ ohms unless otherwise noted. After this, the result of the operation p IMP q will lie in the q memristor.

The IMPLY operation can be extended to implement all Boolean operations by increasing the number of steps and memristors. The area and latency requirements for each Boolean operation in terms of IMPLY resources is shown in Table II. A 0 represents a reset memristor with a high-resistance, or a logical 0 value.

Notice that most of the Boolean operations require more than a single IMPLY operation and more than 2 memristors. Consider the OR operation. First, V_{cond} is applied to P and

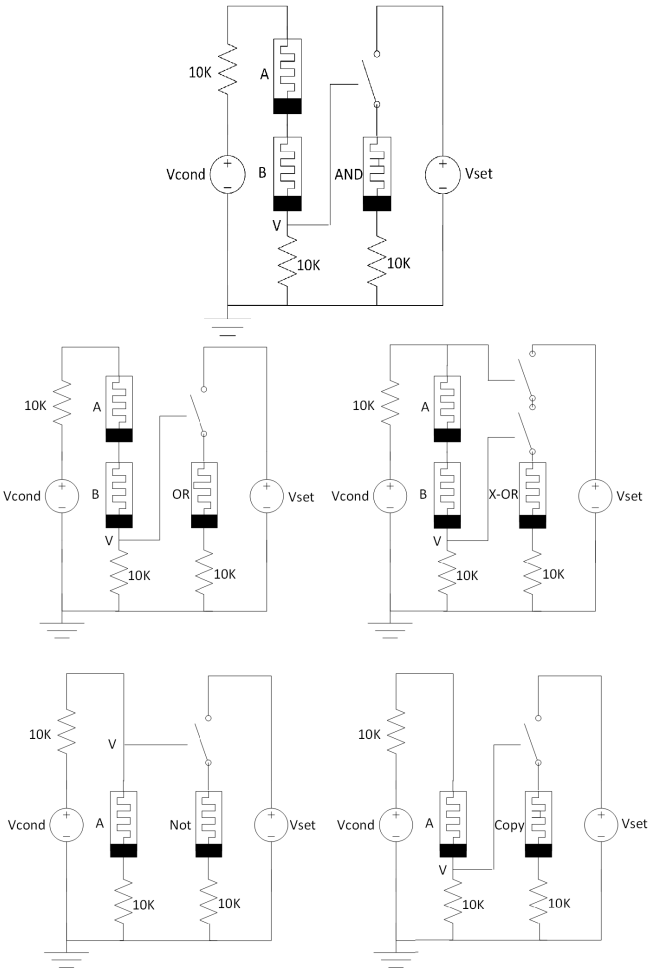

Fig. 3. Circuit for MAD Boolean operations.

V_{set} is applied to a cleared memristor, call it M_c holding the value 0, or high resistance. This executes P IMP 0 and the result lies in M_c. To reset or clear a memristor for IMPLY operations, V_{reset} is applied to the memristor. In this work, $V_{reset} = 5V$. Second, V_{cond} is applied to M_c and V_{set} is applied to the Q memristor to perform (P IMP 0) IMP Q. The result lies in Q. This requires 3 memristors and 2 steps. However, the table shows that the OR operation requires 5 memristors and 4 steps. This is because the value of Q is overwritten in the OR operation. Assuming this value is needed in future operations, the value of Q must be copied, or saved, before Q is overwritten. To perform a COPY, two consecutive NOTs are performed. This copy operation requires two memristors and 2 IMPLY steps, so the total cost of an OR operation is 5 memristors and 4 steps. Similar logic can be applied to obtain the latency and area values for the remaining Boolean operations.

D. MAD Methodology

MAD gates have recently been presented as a similar approach to memristor logic as IMPLY but with lower area and delay [23]. Each MAD gate requires one memristor per operand and a single step to complete. Fig. 3 shows the MAD circuitry for common Boolean operations.

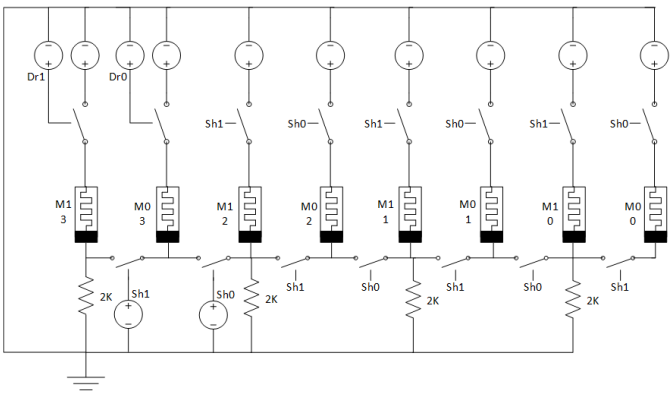

Fig. 4. Schematic for a 4-bit IMPLY shift register.

For Boolean operations that use two input operands, the gate requires two memristors in series, A and B, for the inputs and a switch paired with a third memristor for the result. For operations that use a single operand, such as NOT and COPY, the circuitry is the same except there is only one input memristor A. In both cases, the values of the operands can be preloaded into the input memristors with standard IMPLY set or copy operations.

To perform a Boolean operation, the read voltage V_{cond} is applied across the input memristors. At the same time, the write voltage, V_{set}, is applied to the n-terminal of the result memristor. The application of the V_{cond} and V_{set} voltages are similar to standard IMPLY operations. However, the V_{set} voltage on the output memristor is gated by a switch driven by the voltage sensed from the node V in the input memristor circuit. This switch is identical to the switches found in IMPLY designs, implemented using an ideal transistor or equivalent. The input memristor circuitry essentially acts as a voltage divider - the stored input values in A and B will determine the voltage at node V. If the voltage sensed at node V is greater than the threshold of the switch on the output memristor, the switch will close and the result memristor will be set to a logical 1, else the memristor will remain unchanged. Call the threshold voltage V_{th}. Depending on the Boolean operation, V_{th} will be chosen to correctly implement the desired functionality. Details can be found in prior work [23] that proposes MAD gates.

IV. IMPLY Shift-and-Add Multiplier

An optimized IMPLY-based shift-and-add multiplier is presented that requires $2N^2 + 21N$ steps and 7N+1 memristors. In general, shift-and-add multipliers are lower in area with higher delay as compared to other multipliers. IMPLY-based memristor designs make the same tradeoff. Thus, for designs where shift-and-add multipliers are appropriate, IMPLY is a preferential approach to maintain the same priorities.

First, this work proposes an N-bit shift register implementation for the multiplier which requires 2N memristors and a constant 4 steps for a single shift regardless of the size of N. Fig. 4 shows the schematic for a 4-bit shift register.

A right shift operation can be achieved in the context of IMPLY by performing a COPY operation from each bit i

to bit i-1. Recall that a COPY operation is achieved using IMPLY operations by adding two memristors and performing two consecutive NOT operations. For the first NOT, V_{cond} is applied to the memristor of interest and V_{set} is applied to a cleared memristor M_a. The result ends up in M_a. For the second NOT, V_{cond} is applied to M_a and V_{set} is applied to a second cleared memristor M_b. Now a copied value of the input lies in M_b.

Thus, to perform a shift, this process is performed for each bit i of the register in parallel. Let the shift register be called M and each bit of the shift register be noted as $M1_i$. Based on the description of a COPY operation, another memristor is required per bit to serve as the cleared memristor which will hold the result of the first NOT operation. Call each of these memristors $M0_i$. Now, as described above, a second NOT operation should be performed using $M0_i$ and another cleared memristor. However, in the context of the shift register, the COPY operation stores the result of the second NOT operation for bit i into the M1 memristor in bit i-1. Thus, no more memristors are required. Since all of the bits are performing the same operations at the same time, only two drivers Sh0 and Sh1 are necessary for the entire design, one for all of the M0 memristors and one for the M1 memristors. When Sh1 is driven high, the first NOT operation performs and when Sh0 is driven high, the second NOT operation performs. These switches logically separate each bit, enabling parallel operations with no scalability issues.

Note that an additional 'free' benefit of the IMPLY shift register is the fact that it also produces its inverse during each shift. This value lies in the M0 memristors. Thus, for applications which require the inverse of the operand, this shift register requires no extra memristors or delay. The inverse of the operand can be made available 2 cycles after the value arrives at the shift register.

In the context of a shift-and-add multiplier, the product requires a shift register of size 2N and the multiplier requires a shift register of size N. This equates to a total of 4N memristors and switches plus 2N memristors and switches. Both registers share 6 drivers total, regardless of N, two for V_{cond} and V_{set}, two for the bit connections, and two for the switches which apply V_{cond} and V_{set}. There are also N resistors in the proposed N-bit shift register.

In addition to the shift registers, the shift-and-add multiplier requires an N-bit multiplicand register, a ripple carry adder, and a multiplexer. Prior work presented an optimized N-bit ripple carry adder that requires 7N+1 memristors, 7N drivers, 8N-1 switches, and N resistors [16]. Another prior work that has not yet been published also presented a 2-to-1 N-bit multiplexer that requires 3N+2 memristors, 3N+2 drivers, and 3N+2 switches. An N-bit register requires one memristor, driver, and switch per bit, for a total of N each. Coupling these numbers with the product and multiplier shift registers which together require 6N memristors, 6N switches, and 6 drivers, the total complexity is 17N+3 memristors, 11N+8 drivers, and 18N+1 switches. One iteration of the multiplication consists of a shift, a multiplexer selection, an N-bit addition, and a store operation into the register. As described, the shift operation requires 4 steps - the first to reset the inverse memristors, the second to produce the inverse results (M1-> M0), the third to clear the initial memristors (M1), and the fourth to write the final values (M0-> M1). The N-bit ripple carry adder requires 2N+19 steps, and the multiplexer requires 6 steps. Thus, the initial design of the shift-and-add multiplier requires 2N+29 steps per iteration, or $2N^2 + 29N$ total steps.

TABLE III
SIMPLIFIED IMPLY STEPS FOR SHIFT-AND-ADD MULTIPLEXER

Original Step	Original Goal	Optimized Step	Optimized Goal
Sel imp S0	Not Sel	A imp NotSel	NOT ASel
S0 imp S1	copy Sel	NotSel imp out	ASel + B(NOT Sel)
A imp S0	A NAND Sel		
B imp S1	B NAND Not Sel		
S0 imp out	A AND Sel		
S1 imp out	ASel + B(NOT Sel)		

This work further optimizes the design to reduce this complexity to 11N+6 memristors, 7N+6 drivers, and 13N-1 switches. For a 16-bit adder, this reduces to only 182 memristors - 113 for the adder and multiplexer and 69 for the control and shift registers.

The delay of the multiplier is decreased by 8 steps per iteration as compared to the baseline design. This is equivalent to a 13% reduction in steps each iteration for a 16-bit multiplier. Now, a single iteration requires 2N+21 steps rather than 2N+29, for a total of $2N^2 + 21N$ steps for an N-bit operation. At the same time, the area of the design is also reduced. For a 16-bit multiplier, the area is reduced to less than two-thirds of the original design.

Recall that the optimized ripple carry adder used requires 2N+19 IMPLY steps and the proposed multiplier requires 2N+21 steps per iteration. Thus, the proposed multiplier only adds 2 IMPLY steps per iteration for performing the shift register, multiplexer, and store operations.

Many optimizations have been performed to reduce the delay and area. First, the multiplier register has been incorporated into the shift register to eliminate the N-bit multiplier register entirely. The multiplicand register has also been eliminated. Since the multiplicand is the only possible operand to be added to the product register in each iteration, it can permanently lie in the A operands of the adder. This also saves 2 steps for performing a copy from the shift register to the adder during each iteration of the multiplication. Thirdly, the step that resets the M0 memristors of the shift register can be performed during the ripple carry addition, reducing the observed overhead for the shift from 4 to 3. Lastly, the IMPLY-based multiplexer can be simplified since one of the inputs is always 0. Given this knowledge, the multiplexer from prior work can be reduced to require a single memristor and 2 steps. This sequence can be seen in Table III.

Since input B is 0, the second term in the multiplexer equation Out=ASel + B(NOT Sel) is 0 too. The equation simplifies to Out=ASel which is a simple AND operation. Additionally, the design logically separates each bit of the N-bit multiplexer so that the bits can perform in parallel. Each bit can sense the Sel and inverse Sel signal independently, and perform the two IMPLY steps in lockstep. Now the entire multiplexer operation requires only 2 steps.

TABLE IV
EXECUTION STEPS FOR A SINGLE CYCLE OF A SHIFT-AND-ADD MULTIPLIER FOR BIT 0

STEP	FUNCTIONALITY	GOAL
1	NOT Sel	Sel ->M2
2	NOT ASel	A->M2
3	A (mux result)	M2->M5
4	B imp 0	Product Reg->M3
5	B	M3->M1
6	(A imp 0) imp (B imp 0)	M2->M3
7	A imp B	M5->M1
8	((A imp 0) imp (B imp 0)) imp 0	M3->M0
9	B imp (A imp 0)	Product Reg->M2
10	B imp (A imp 0) imp 0	M2->M4
11	A XOR B	M1->M0
12		FALSE(M1,M2,M3)
13	(A XOR B) imp 0	M0->M2
14	Cinimp ((A XOR B) imp 0)	Cin->M2
15	CinAND (A XOR B) + AB->COUT	M2->M4
16		NEXT BIT IS READING CIN
17	Cinimp 0	Cin->M1
18	C+ (A XOR B)	M1->M0
19		NEXT BIT IS READING CIN
20	(C+ (A XOR B)) imp 0	M0->M3
21	(C NAND (AXORB)) imp ((C+ (A XOR B)) imp 0)	M2->M3
22	(C NAND (AXORB)) imp ((C+ (A XOR B)) imp 0) imp 0 =SUM	M3->Zeroed out Product reg
23		FALSE ALL except input

Again, the steps for the multiplexer can be incorporated into the first steps of the standard ripple carry addition. The two optimizations described are coupled with strategic Boolean manipulation and rearrangement based on IMPLY properties to produce an optimized design. Specifically, the first 3 steps of the standard ripple carry adder translate into 4 steps in the shift-and-add multiplier in order to incorporate both the shift and multiplexer functionality. The resultant series of steps is shown in Table IV for the first bit of the multiplication.

After the result of the addition is stored in the product register, the memristors in the adder and the memristors holding the inverse operand in the shift registers are reset in order to prepare for the next iteration of the multiplication. This is necessary for proper IMPLY functionality. However, each bit can reset as soon as its computation is complete. For example, after bit b propagates its carry-out value and resolves its sum, it can reset its intermediate memristors while bit b+1 performs its remaining computation. Similarly, once the sum value is sensed into the shift register, all of the sum memristors can be cleared while the shift is occurring.

A. IMPLY-Specific Optimizations

Although this design is highly optimized, it can still be significantly improved by rethinking its implementation at the top-level based on the characteristics of memristor-based logic.

The IMPLY operation is based around the notion of strategic voltage applications to the terminals of memristors in a serial fashion. This essentially represents "bringing the computation to the data," a popular approach to moving away from traditional Von Neumann semantics. The data lies in the memristors and the computation, dictated by driven voltages, is performed on the memristors. This knowledge can be leveraged to remove the need for the shift registers entirely from the multiplier design. In traditional CMOS, the shift register is used to physically move the bits to their appropriate bit indices before performing the next iteration. In the memristor domain, this can be achieved by virtually shifting the driver circuitry across the bits. As a simple example, consider an 8-bit multiplication. Let the 2N-bit shift register P begin such that $P_i[15:8] = 0$ (for the initial product) and $P_i[7:0]$ = multiplier B[7:0]. The first addition would take $P_i[15:8]$ and add either the multiplicand or 0 based on $P_i[0]$, store the result back into $P_i[15:8]$, and then shift the shift register once to the right.

Instead, one modification will be made to the format of the shift register. Rather than storing $P_i[7:0]$ = multiplier B[7:0], $P_i[7:0]$ = multiplier B[0:7] is stored. Now, bit 0 of the multiplier will be sensed from P_i [7]. Rather than storing the final sum into $P_i[15:8]$, the drivers can store the result into $P_i[14:7]$, achieving the shift concurrently with the sum. This will overwrite the least significant of the multiplier B, maintaining B[1:7], just as the shift would have. The second iteration can now proceed identically as before, again taking $P_i[15:8]$, without the need for a shift, and sensing P_i [6] for the multiplier bit. Thus, there is no need for the shift register at all. Instead, a simple 2N-bit register is necessary for holding the intermediate multiplier and product values at each iteration. This removes 3 steps from each iteration (for the shift operation) and 2N memristors (for the inverse shift values).

However, by the same logic, the area and delay can be further reduced. Consider two consecutive iterations of an 8-bit multiplication, i and $i+1$. Let the sum in the first iteration be $S_i[7:0]$. $S_i[7:0]$ is computed and stored into the 16-bit product register P such that $P_i[14:7] = S_i[7:0]$ and $P_i[6:0] = B[1:7]$. Then, iteration $i+1$ begins and bits $P_i[15:8]$ (essentially $\{0,S_i[7:1]\}$ are loaded into the ripple carry adder). Note that all of these steps are unnecessary—result of one iteration of the addition is essentially inserted directly back into the adder as an operand in the subsequent iteration. Thanks to the driver-based nature of IMPLY logic, this can be achieved directly in the final stage of iteration i. Rather than copying $Sum_i[7:0]$ from the adder into $P_i[15:8]$, the Sum results can be directly resolved into their operand location for iteration $i+1$. In other words, for a bit b in the ripple carry adder, when Sum_b is being computed, rather than applying V_{set} to the Sum memristor for bit b, V_{set} is applied to the B operand in bit b-1. Now, at the end of iteration i, $\{0,S_i[7:1]\}$ lies in the B operand. Iteration $i+1$ can begin as soon as the intermediate memristors are reinitialized to 0. This optimization is only possible due to the benefits of the IMPLY operation. It removes the need for the product result memristors and the sum memristors internal to the ripple carry adder since the sums are stored directly into the B operands. Thus, the only required circuitry is the modified N-bit ripple carry adder and the N-bit multiplier register for a total of 7N+1 memristors, 7N drivers, 8N-1 switches, and $N+1$ resistors. The fully optimized schematic for an 8-bit multiplier is shown in Fig. 5.

The top two columns of the schematic are the ripple carry adder. This adder consists of 8 full adders, each of which produces a sum output and feeds the carry out signal to the next full adder. Each full adder is connected to the input from the previous adder in the chain by a driver. This driver selects the result carry from the previous bit in the adder once it has

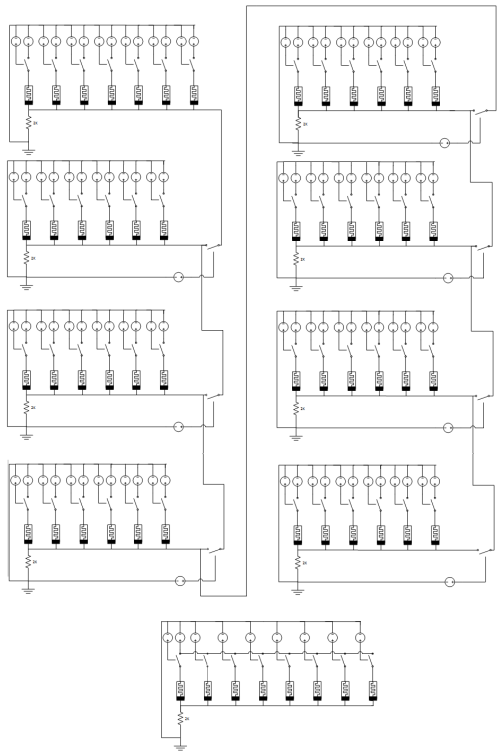

Fig. 5. 8-bit shift-and-add IMPLY multiplier.

TABLE V
FINAL EXECUTION STEPS FOR THE SHIFT-AND-ADD MULTIPLIER OPTIMIZED FOR IMPLY

11		FALSE(M1,M2,M3,B)
...	...	...
22	(C NAND (AXORB)) imp ((C+ (A XOR B)) imp 0) imp 0 =SUM	M3->B_{i-1}
23		FALSE ALL except inputs

been resolved. The use of these drivers creates a structural separation, effectively placing each basic block of logic on its own row. This allows each full adder to operate in parallel without interference across the full adders. Within a full adder, only one IMPLY operation (or two memristors) are active in a step and each adder has its own single load resistor. Similarly, in the shift register, each shift is isolated through the use of switches so that each IMPLY operation is independent of the others. These design decisions eliminate any unwanted interference on the output signals and allow the design to have any width without scalability issues.

The bottom row of Fig. 5 implements the multiplier register. Each memristor in the register is logically disjoint so that it can be sensed in its particular iteration. The delay is maintained at $2N^2 + 21N$ but the modified steps of execution are shown in Table V.

The B memristor is cleared in step 11 along with other intermediate results since its value is never used after step 8. This removes the need to clear the memristor before storing the sum result into it in step 22. A simulation waveform for the execution of an 8-bit shift-and-add multiplier for the multiplication $0\times AA \times 3 = 0\times FE$ is shown in Fig. 6.

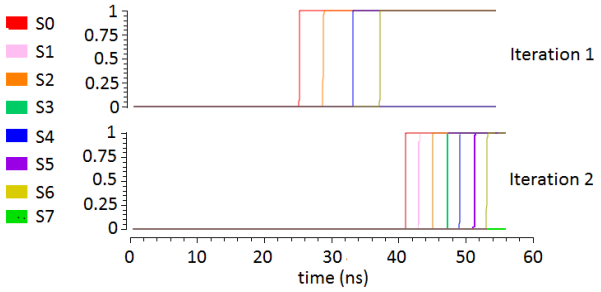

Fig. 6. Waveform for $0\times AA$ times $3 = 0 \times 1FE$ after two iterations of the multiply.

The waveform shows the state of simulation after the first two iterations of the multiply. In the first iteration, the multiplier bit 0 is sensed as 1 and $0\times AA$ is added to 0. The result is $0\times AA$ but since the sum is shifted as part of the addition, the sum bits $S[7:0] = 0\times AA >> 1 = 0\times 55$. Bit 0 is resolved at t = 23n and each consecutive bit is resolved every 2 steps after. Every other bit resolves to 1 as shown. In the second iteration at t = 40ns, the multiplier bit 1 is sensed as 1 and the product register 0×55 is added to $0\times AA$. The sum, $0\times FF$ is shifted and $S[7:0] = 0\times FF >> 1 = 0 \times 7F$.

1) Pipelining:: Lastly, the proposed shift-and-add multiplier is optimized to overlap individual iterations of the multiplication. Again, due to the fact that IMPLY operations essentially bring the computation to the data rather than vice-versa, the bits of the adder can be individually driven. As soon as bit b completes its first addition iteration, it propagates its carry-out, stores the sum result in bit b-1 in the B operand memristor as discussed, and resets its memristors. At this point, the full adder is able to begin a second iteration of the multiplication by sensing the next bit of the multiplier register. It can do this independent of the other bits in the adder, which are still performing the first iteration. With this functionality, it is possible to begin a new iteration of the multiplication every 24 cycles. The addition itself takes 23 steps and the sum value from the next bit can be received in the following step. Thus, if the multiplier is fully utilized, it can perform a multiplication every 24N steps. This reduces the delay complexity from $O(N^2)$ to O(N).

To perform pipelining correctly, the multiplier register must be modified. In the non-pipelined implementation, bit b_i in the multiplier register is read by each consecutive bit in the ripple carry adder during iteration i. In the current multiplier register, V_{cond} is only applied to one memristor in the register at a time. However, if the design is changed to pipeline additions, different bits in the ripple carry adder will be on different iterations of the adder and will potentially need access to different bits in the multiplier register during the same iteration. Thus the drivers on the multiplier register have been strategically timed so that no two bits of the multiplier register are needed during the same step. In this way, the circuitry can remain identical and only the driver logic needs to change to facilitate pipelining.

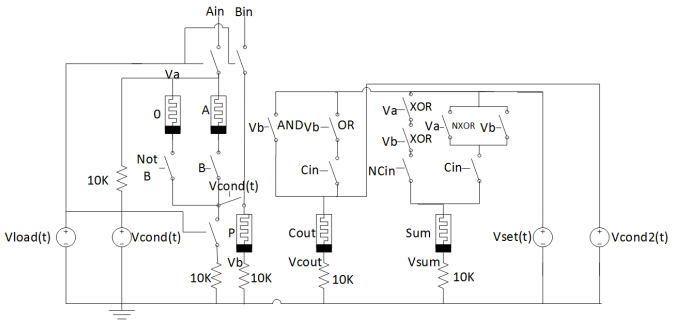

Fig. 7. Full adder for the proposed shift-and-add multiplier.

V. MAD Shift-and-Add Multiplier

Although it is possible to implement shift-and-add multipliers using IMPLY operations, it requires heavy optimization to achieve comparable performance to CMOS. The proposed IMPLY design has been optimized and parallelized to require only 2 more steps than the IMPLY ripple carry adder to perform the shift and multiplexer logic. Unfortunately, this still results in 2N+21 IMPLY steps per iteration of the multiplier, 2 fewer steps per iteration than the traditional CMOS design.

To overcome these limitations, a new design for a shift-and-add memristor-based multiplier based on MAD gates is proposed. First, the proposed design minimizes the delay internal to each full adder in the ripple carry adder. This design uses an optimized full adder from prior work as a baseline [23]. The baseline requires 4 memristors, 5 resistors, 13 switches, and 4 drivers and only 1 step once the inputs are initialized. This work further improves on this by leveraging information about the shift-and-add multiplier context. The resultant full adder is shown in Fig. 7.

In a shift-and-add multiplier, both of the '0' and 'A' inputs to the multiplexer are known and constant throughout the iterations. Thus, rather than having a single input A memristor like the baseline full adder, this adder has two memristors to hold the value of the multiplicand (A) and the value 0. These represent the two inputs to the multiplexer in the multiplier. They can be loaded once at the beginning of the multiplication and held resident in the full adders for the entire multiplication. This optimization removes the need for the multiplicand register entirely since the multiplicand now lies constant in the full adders. This also eliminates the overhead of copying the multiplicand operand into the ripple carry adder during each iteration. Thus, both area and latency are improved with the addition of only a single memristor and switch into the design. The P memristor represents the bit operand of the running product register.

At initialization, the V_{load} signal is driven high and the value of the multiplicand operand A is set by A_{in} and the value of the product register is set by B_{in}. Then, the multiplier and carry-in bit are both sensed simultaneously to resolve the full adder in a single step. The multiplier bit B and its inverse, NOT B, are sensed by applying a read voltage, V_{cond}, to the multiplier register B bit i. The sensed voltages are used as the drivers on the switches labeled 'NOT B' and 'B' to select either the multiplicand, A, or the 0 memristor for the addition. This incorporates the multiplexer functionality into the adder without any delay or area requirements.

At this time, V_{cond} is applied to connect the selected input A and the product memristor in series to ground. V_{cond} is also applied to the carry-in bit in the previous adder (indicated by the value of C_{in}). The voltages at V_a and V_b are sensed as voltage division signals to drive the gates that resolve the final carry-out and sum memristors. This mimics traditional MAD gate behavior. The gates are labeled 'AND' and 'OR' to indicate the Boolean operation achieved by the given threshold voltage of that gate. For example, the AND switch only closes when the voltage sensed at node V_b is greater than the threshold voltage denoting that both inputs are '1'. Similarly, the OR switch only closes when the voltage sensed at V_b is greater than the threshold voltage denoting that at least one input is a '1'. Thus, the leftmost path from V_{set} to Cout implements (A AND B) and the rightmost path implements (A OR B) AND C_{in}. As a result, V_{set} is gated to Cout iff (A AND B) OR ((A OR B) AND C_{in}). The sum is computed as NC_{in}(A XOR B) OR C_{in}(A NXOR B). Thus, the two paths from V_{set} to the sum memristor can be implemented as shown using the carry-in information, the XOR gate from Fig. 3, and its inverse. Threshold voltages on the V_a and V_b switches are selected accordingly. In all, the full adder only requires a single step after initialization of the inputs. The inputs and carry-in signals are all sensed at the same time that the sum and carry-out memristors are set.

The full adder design can be extended to an N-bit ripple carry adder by replicating the circuit N times and connecting each bit with driver logic as shown in prior work [23]. Essentially, each adder needs to be able to sense or read the carry-out value from the previous full adder to use as its carry-in. This can be done using a traditional MAD read operation, as is done for the multiplier bits.

A similar optimization as was done for the IMPLY design is performed by leveraging the driver nature of MAD gates. The design will largely remain unchanged except that the logic shown in Fig. 7 for the sum memristor, will now be resolving sum_{n-1}. Now, the sensing behavior is identical for the full adder, but rather than driving the local bit i's sum memristor with the signals, it will drive the sum memristor in the previous consecutive bit, i-1. In other words, rather than applying V_{set} to the sum memristor in full adder i, V_{set} is applied to the sum memristor in full adder i-1. This is possible because the sum memristor and its drivers are completely independent from the rest of the circuit. Since, the result of each full adder's sum calculation is resolved into the previous full adder, the shift functionality is achieved. This optimization removes the need for the product shift register and shift delay from the design.

However, with the updated circuit, the B operand for the next iteration will lie in the sum memristor rather than the P memristor, posing an issue for the next iteration. Thus, the sum memristor will simply be mapped to the P memristor. In other words, the driver and switch logic shown for the sum memristors will actually be performed on the P memristor. Cumulatively, the full adder is modified to store into the previous bit's P memristor rather than its own sum memristor but none of the underlying logic changes. This removes

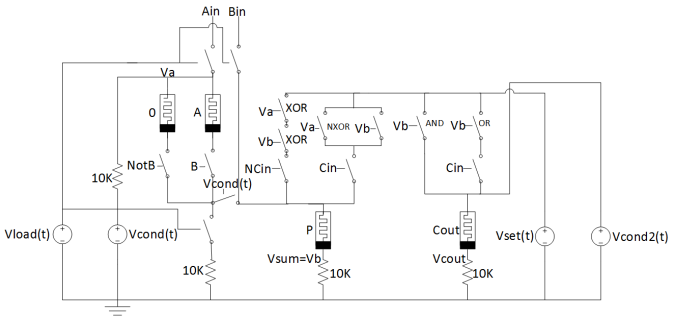

Fig. 8. Optimized full adder for the proposed shift-and-add multiplier.

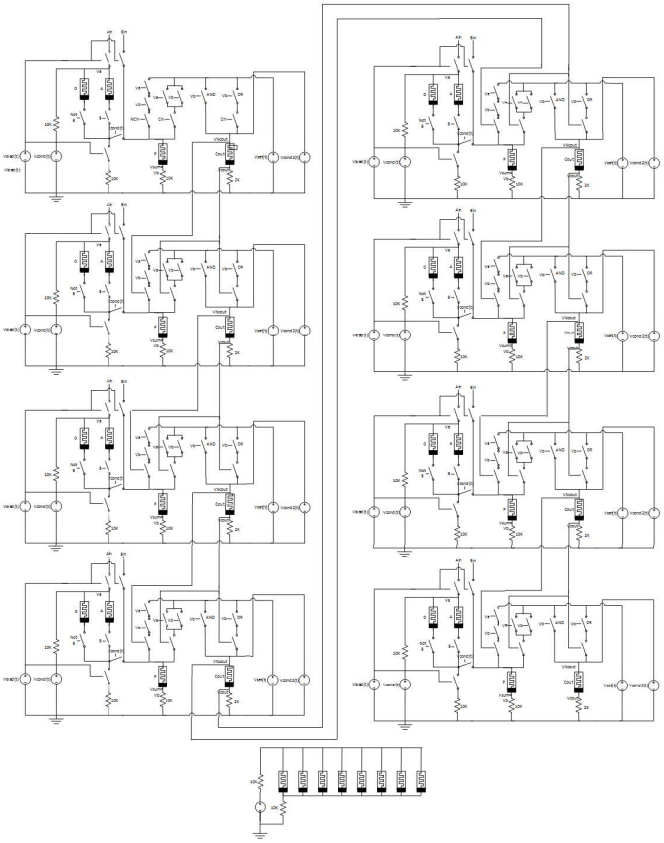

Fig. 9. 8-bit MAD shift-and-add multiplier.

another N memristors by eliminating the sum memristors. The modified MAD full adder is shown in Fig. 8.

Now, the design only requires N replications of the modified full adder and an N-bit register for the multiplier. Each multiplier bit will be sensed one iteration at a time by the drivers, to logically perform a shift without physically performing one. Thus, the total complexity of the design is 5N memristors, 3N+2 drivers, and 14N switches. The full schematic for an 8-bit implementation can be seen in Fig. 9.

The key takeaway from this figure is the connectivity between the full adders. The figure explicitly shows the connection between the carry-out signal of a full adder and the carry-in signal of the next. The carry-out memristor is probed at both of the terminals to drive the gate logic on the subsequent adder.

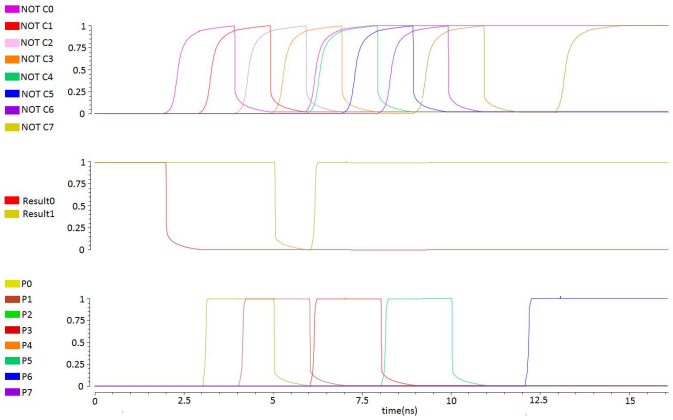

Fig. 10. Waveform for overlapping additions in the proposed shift-and-add multiplier for $0 \times 56 \times 3$.

The delay of a single iteration consists of the delay of the first full adder +1 step per consecutive bit for carry propagation. Thus, the total delay of an addition is $N+1$ steps and the total delay for the multiplication is $N^2 + N$ steps.

The MAD implementation can also be modified to accommodate pipelined additions. In the same manner as explained for the IMPLY implementation, each bit can begin the next iteration of the addition as soon as it propagates its carry-out and sum results to the next bit. In the next cycle, it receives the sum result from the next bit into its P memristor, and it resets its carry-out and product memristors. Now it can start the next iteration. This implies that a new addition can occur every 4 cycles. Consider bit b. When the carry-in is ready, the full adder is able to set its carry-out memristor and the previous full adder's sum memristor. In the second cycle, the full adder resets the product register. In the third cycle, bit b+1 reads bit b's carry-out memristor and resolves its sum into the P memristor in bit b. In the fourth step, bit b resets the carry-out memristor. The total effective latency of the MAD shift-and-add multiplier is now 4N steps rather than $N^2 + N$. Like the IMPLY implementation, each memristor in the multiplier register must be logically separated to function properly, adding N-1 resistors and a driver.

A waveform showing this ability for $0 \times 56 \times 3$ is shown in Fig. 10.

The waveform shows two full iterations of the multiplication. In the first iteration, 0×56 is added to 0. The sum and inverse carry out memristors are each resolved 1 step after the previous bit from $t = 2$ns to $t = 9$ns. The inverse carry out values all resolve to 1, one cycle at a time, and the sum memristors resolve to the A inputs. Recall that the shift happens as part of this action, so the P memristors store $0 \times 2B$ rather than 0×56. The shifted-out 0 is stored into the LSB of the storage product register, Result0.

Bit 0 resolves its carry-out and sum at $t = 2$ns. At $t = 3$ns, the sum memristor in bit 0 is set. At $t = 4$ns, the carry-out memristor is cleared. At $t = 5$ns, the second iteration begins, adding $P = 0 \times 2B$ to the multiplicand A. This results in all of the inverse-carry-outs going high and the p memristors to 0×81. Given the shifted sum result, the product register has 0×40 at the end of the second iteration and bit 1 of the

TABLE VI
DELAY AND AREA COMPARISONS OF THE PROPOSED IMPLY SHIFT-AND-ADD MULTIPLIERS

	Baseline	Proposed-Optimized Shift Register	Proposed-No Shift Register
Delay	$2N^2$+29N	$2N^2$+21N	$2N^2$+21N
Area	17N+3 memristors 11N+8 drivers 18N+1 switches	11N+6 memristors 7N+6 drivers 13N-1 switches	7N+1 memristors 7N drivers 8N-1 switches

TABLE VII
DELAY AND AREA COMPARISONS OF THE PROPOSED SHIFT-AND-ADD MULTIPLIERS

	CMOS	Proposed IMPLY	Proposed MAD
Delay	$2N^2$+23N	$2N^2$+21N	N^2+N
Pipelined Delay	$2N^2$+23N	24N	4N
Area	132N+6 MOSFETs	7N+1 memristors 7N drivers 8N-1 switches	5N memristors 3N+2 drivers 14N switches

storage product register has a 1. Two complete iterations of the multiplication complete at t = 13ns.

VI. SHIFT-AND-ADD MULTIPLIER COMPARISON AND DISCUSSION

The proposed shift-and-add multipliers significantly optimize the delay and area of the traditional CMOS design.

The IMPLY-based multiplier went through numerous iterations to fully leverage all of the optimizations that could be extracted from the traditional shift-and-add design. A summary of the baseline and iterative designs' costs are shown in Table VI.

The optimizations made to incorporate the multiplier into the product register and combine the multiplexer logic into the ripple carry adder effectively removed 6N-3 memristors, 4N+2 drivers, and 5N+2 switches from the design, reducing the componential count by 32%. Not only is this a significant savings in terms of area, but 8N steps were shaved off the total delay, too. Once the design leveraged IMPLY characteristics and did not conform to the traditional shift-and-add design, the area and latency improved by even more. The final design requires essentially the same area as the IMPLY ripple carry adder proposed in prior work, yet is capable of performing all the shift-and-add multiplier logic (the multiplexer, the operand registers, and the shift registers). This is less than half the number of components used in the baseline while still reducing the delay by 8N steps.

Although the IMPLY design is highly optimized, the MAD implementation still offers a lower area, lower delay option. A full comparison of the delay and area of each design and their CMOS counterparts is given in Table VII.

The number of memristors and drivers reduces by 43% as compared to the optimized IMPLY design. This is a total savings of 75% as compared to the original baseline IMPLY implementation. This comes at the cost of more switches as compared to IMPLY, but the number of switches is still fewer than the baseline. Both of the designs significantly reduce the number of components to about 1/6 of the CMOS design. Note that this is not a complete comparison since memristors are smaller area than CMOS components. The proposed designs provide a two-fold improvement in area, not only reducing the number of components, but the size of the components too.

The MAD design also improves the latency of the multiplication to only $N^2 + N$ steps. This is less than half as many steps as the optimized IMPLY and CMOS designs. When the designs are fully utilized, both benefit from the ability to pipeline consecutive additions. The traditional CMOS design cannot perform pipelining, thus its latency remains the same. For both proposed designs, the latency reduces from $O(N^2)$ to O(N). For the IMPLY implementation, the effective latency of an N-bit multiplication reduces from $2N^2 + 21N$ steps to 24N. For a 16-bit multiplication, this is equivalent to 848 steps versus 384 steps, which is more than a 2× improvement. For the MAD implementation, the effective latency is even less, only 4N steps per multiplication - 64 steps for an entire 16-bit multiplication. This is a very significant improvement over traditional CMOS designs which require 880 steps. Although we can compare step counts between the memristor and CMOS designs, it is not possible to perform a time-delay comparison. As mentioned, depending on the voltage applied to the memristors, their switching times can vary and will likely not coincide with CMOS.

This work does not consider the power or energy consumption of the previous or proposed designs. Although this is an important metric, the infancy of memristor models makes it difficult to get accurate power or energy measurements with high confidence from simulation. However, prior work has shown that an IMPLY Boolean operation requires between 3.4e-13J and 3e-12J and a single MAD operation requires 3e-14J [23]. From this, it is likely that both implementations would offer competitive designs in terms of energy consumption, with MAD gates requiring an order of magniture less energy than IMPLY.

VII. EXTENSION TO BOOTH MULTIPLIERS

Both the IMPLY and MAD designs can be extended to implement a Booth multiplier. These designs assume that the value for the 2's complement of the multiplicand is calculated and initialized into the design *a priori*. This can be done using a standard ripple carry adder.

A. IMPLY Booth Multiplier

The only change necessary to enhance the IMPLY shift-and-add multiplier design to Booth is the multiplexer. The multiplexer requires a 2-bit select line for the two bits of the multiplier that are read each iteration. The 2-bit multiplexer can be simplified since two of the inputs are always 0. In the Booth design, when the select lines are 'b00 or 'b11, the input is 0 so the equation for the multiplexer simplifies to $A_i\overline{b_i}b_{i-1} + A_i'b_i\overline{b_{i-1}}$, where A_i' represents the 2's complement of A for bit i.

The multiplexer executes the steps in Table VIII. Let b_1 and b_0 represent the two memristors of interest from the multiplier register.

This multiplexer has been simplified and optimized by rearranging the IMPLY operations to minimize the number

TABLE VIII
MODIFIED STEPS FOR A 2-BIT IMPLY MULTIPLEXER FOR A BOOTH MULTIPLIER

Num	Step	Goal
1	b_0 imp T_0	$\overline{b_0}$
2	T_0 imp M_5	copy b_0
3	b_1 imp M_5	NAND $b_1\overline{b_0}$
4	b_1 imp M_2	$\overline{b_1}$
5	M_2 imp T_0	NAND $\overline{b_1}b_0$
6	$\overline{A}$ imp M_5	NAND $\overline{A}b_1\overline{b_0}$
7	A imp T_0	NAND $A\overline{b_1}b_0$
8	T_0 imp T_1	$A\overline{b_1}b_0$
9	T_1 imp M_5	$A\overline{b_1}b_0 + \overline{A}b_1\overline{b_0}$

of steps. Also, the steps that only rely on values from the multiplier register (and not on the input operands) in iteration K can be performed in iteration K-1 during the addition process (after the multiplier register has been read for iteration K-1). The multiplexer is optimized to execute these steps first to reduce the effective latency of each iteration. Note that steps 1-5 in Table VIII can all occur during the execution of the previous addition iteration so only 4 steps are on the critical path. These 4 steps replace the original 3 steps in Table IV and the multiplexer result exists in M5 as before. Thus, Booth's multiplier only takes a single step more per iteration than the shift-and-add multiplier. Two additional memristors, T_0 and T_1, are required to handle the additional multiplexer complexity.

The only other modification to the multiplier design is that the drivers on the multiplier register must change to sense two bits in each iteration i, b_i and b_{i-1}, rather than 1. This does not change the circuitry, but it does change the application of the voltage signals for each bit in the multiplier.

B. MAD Booth Multiplier

To accommodate Booth's algorithm in the MAD multiplier design, changes must be made to the multiplexer logic. A third memristor that holds the 2's complement of the multiplicand will be added in series with the multiplicand and '0' memristors to serve as a third operand to the multiplexer. The logic that selects between these potential operands also changes. Let M_i be the bit in the multiplier that selects the input operands in the current iteration. Originally, the value of M_i was used to select either 0 (if M_i is 0) or the multiplicand (if M_i is 1). This was done by placing switches on each of the memristor input operands, each gated by M_i and $\overline{M_i}$. This must be modified to accommodate checking the value of two multiplier bits, M_i and M_{i-1}. Now, if $\{M_i,M_{i-1}\}$ == b'00 or $\{M_i,M_{i-1}\}$ == b'11, '0' is selected, if $\{M_i, M_{i-1}\}$ == b'01 the multiplicand is selected, and if $\{M_i,M_{i-1}\}$ == b'10 then the 2's complement of the multiplicand is selected. Fig. 11 shows the modified adder for the Booth multiplier.

The MAD Booth full adder requires N additional memristors and 6N switches but has the same delay as the original full adder - just 1 step after initialization. Note that just as for the IMPLY design, the drivers on the multiplier register must change to sense two bits in each iteration i, b_i and b_{i-1}, rather than 1.

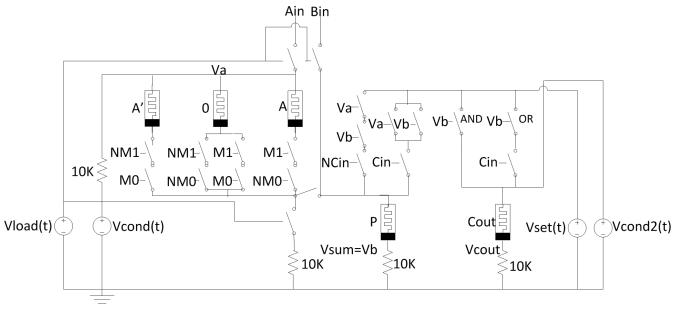

Fig. 11. MAD full adder for a booth multiplier.

TABLE IX
DELAY AND AREA COMPARISONS OF THE PROPOSED BOOTH MULTIPLIERS

	CMOS	Proposed IMPLY	Proposed MAD
Delay	$2N^2$+26N	$2N^2$+22N	N^2+N
Amortized Delay	$2N^2$+26N	25N	4N
Area	166N+8 MOSFETs	9N+1 memristors 9N+1 drivers 10N-1 switches	6N memristors 3N+3 drivers 20N switches

C. Booth Multiplier Comparisons

A comparison of the proposed Booth multipliers against the traditional CMOS design is given in Table IX. The same limitations on the comparison still exist as discussed for the shift-and-add multipliers.

The IMPLY design has been optimized to favor latency over area. As a result, the IMPLY based design requires 4 fewer steps than the traditional CMOS design per iteration. Each bit of the adder needs 2 more memristors and their drivers to implement the 2-bit multiplexer. It is possible to lower the area by reusing memristors from the original design rather than introducing the new memristors T_0 and T_1. However, this will increase the latency. There is also one more driver introduced to enable sensing two bits at once from the multiplier register. The total complexity for an N-bit IMPLY Booth multiplier is 9N+1 memristors, 9N+1 drivers, and 10N-1 switches. This is less than 20% the number of components that CMOS requires which is 166N+8 MOSFETs. It can also be pipelined to require a total of only 25N steps.

The N-bit MAD Booth multiplier requires 6N memristors, 3N+3 drivers, and 20N switches and $N^2 + N$ steps. This is a nearly identical component count to the IMPLY design and also less than 20% of the number of components for the CMOS design. The number of steps reduces to $N^2 + N$ steps which is less than 30% of the number of CMOS steps and less than 50% the number of IMPLY. It can also be pipelined by the same logic as the shift-and-add multiplier to only require an effective latency of 4N steps per multiplication. This is significantly faster than CMOS and IMPLY, reducing the delay from 928 steps in CMOS and 400 in IMPLY to only 64 steps in MAD.

VIII. CONCLUSION

Recently, memristors have emerged to the forefront as a new circuit element. However, minimal work has explored the

application of these devices to complex systems, especially in the context of arithmetic units. The majority of prior work that does exist has focused on basic Boolean operations and small circuits or has presented purely theoretical, unoptimized designs. No prior work has explored the application of memristors to multipliers - a critical component in all modern system architectures.

This work presents two designs for shift-and-add multipliers—one using IMPLY operations and the other using MAD methodologies. Both works are heavily optimized to achieve latency and delay improvements over the traditional CMOS design. Because of the driver-based nature of both approaches, the shift register can be effectively removed from the multiplier design entirely. Additionally, the multiplexer functionality can be achieved more efficiently by incorporating the logic into the adders themselves in the designs. These optimizations reduce the IMPLY design's area from 17N+3 memristors to only 7N+1 memristors and the accompanying driver circuitry. This is equivalent to the numbers of memristors required for the optimized IMPLY ripple carry adder alone in previous work. The MAD design requires even less area at only 5N memristors and their drivers and switches. Both designs reduce the number of components to 1/6 of the traditional CMOS design.

The delay of the IMPLY design is also reduced, decreasing from $2N^2 + 29N$ to $2N^2 + 21N$ steps. Each addition iteration of the multiplication only requires two extra steps to perform the multiplexer, shift, and selection functionality of the multiplier. This is fewer steps than $2N^2 + 23N$ required by the CMOS design. The MAD implementation further decreases the latency to less than half of this delay to $N^2 + N$ steps.

Both proposed designs can be leveraged further to pipeline consecutive additions. This improves the delay for a multiplication to just 24N and 4N for the IMPLY and MAD designs respectively. This is a significant improvement over CMOS technologies which cannot be pipelined, reducing the delay from $O(N^2)$ to O(N).

The proposed shift-and-add designs are modified to implement Booth multipliers. For the IMPLY-based approach, this requires an additional 2N memristors and only increases the pipelined latency for a multiplication from by N steps. For the MAD-based approach, the delay remains the same as the shift-and-add multiplier and the complexity only increases to 6N memristors, 3N+3 drivers, and 20N switches. Thus both proposed designs offer competitive, lower area designs than the CMOS equivalent.

REFERENCES

[1] L. O. Chua, "Memristor—The missing circuit element," *IEEE Trans. Circuit Theory*, vol. CT-18, no. 5, pp. 507–519, Sep. 1971.

[2] Y. Zhang, Y. Shen, X. Wang, and Y. Guo, "A novel design for a memristor-based OR gate," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 62, no. 8, pp. 781–785, Aug. 2015.

[3] Y. Zhang, Y. Shen, X. Wang, and L. Cao, "A novel design for memristor-based logic switch and crossbar circuits," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 62, no. 5, pp. 1402–1411, May 2015.

[4] X. Zhu, X. Yang, C. Wu, N. Xiao, J. Wu, and X. Yi, "Performing stateful logic on memristor memory," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 60, no. 10, pp. 682–686, Oct. 2013.

[5] S. Kvatinsky *et al.*, "MAGIC—Memristor-aided LoGIC," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 61, no. 11, pp. 895–899, Nov. 2014.

[6] Y. Yang, J. Mathew, S. Pontarelli, M. Ottavi, and D. K. Pradhan, "Complementary resistive switch-based arithmetic logic implementations using material implication," *IEEE Trans. Nanotechnol.*, vol. 15, no. 1, pp. 94–108, Jan. 2016.

[7] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, pp. 80–83, May 2008.

[8] K. Bickerstaff and E. E. Swartzlander, "Memristor-based arithmetic," in *Proc. Conf. Rec. 44th Asilomar Conf. Signals, Syst. Comput.*, Nov. 2010, pp. 1173–1177.

[9] D. Mahajan, M. Musaddiq, and E. E. Swartzlander, "Memristor based adders," in *Proc. 48th Asilomar Conf. Signals, Syst. Comput.*, Nov. 2014, pp. 1256–1260.

[10] A. H. Shaltoot and A. H. Madian, "Memristor based carry lookahead adder architectures," in *Proc. IEEE 55th Int. Midwest Symp. Circuits Syst. (MWSCAS)*, Aug. 2012, pp. 298–301.

[11] M. Teimoory, A. Amirsoleimani, J. Shamsi, A. Ahmadi, S. Alirezaee, and M. Ahmadi, "Optimized implementation of memristor-based full adder by material implication logic," in *Proc. 21st IEEE Int. Conf. Electron., Circuits Syst. (ICECS)*, Dec. 2014, pp. 562–565.

[12] M. Teimoory, A. Amirsoleimani, A. Ahmadi, S. Alirezaee, S. Salimpour, and M. Ahmadi, "Memristor-based linear feedback shift register based on material implication logic," in *Proc. 22nd Eur. Conf. Circuit Theory Design (ECCTD)*, Aug. 2015, pp. 1–4.

[13] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Memristor-based material implication (IMPLY) logic: Design principles and methodologies," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 10, pp. 2054–2066, Oct. 2014.

[14] S. Kvatinsky, N. Wald, G. Satat, A. Kolodny, U. C. Weiser, and E. G. Friedman, "MRL—Memristor ratioed logic," in *Proc. 13th Int. Workshop Cellular Nanosc. Netw. Appl. (CNNA)*, Aug. 2012, pp. 1–6.

[15] T. Singh. (Jun. 2015). "Hybrid memristor-CMOS (MeMOS) based logic gates and adder circuits." [Online]. Available: http://arxiv.org/abs/1506.06735

[16] L. Guckert and E. E. Swartzlander, "Optimized memristor-based ripple carry adders," in *Proc. 50th Anniv. Asilomar Conf. Signals, Syst., Comput.*, to be published.

[17] G. Rose, J. Rajendran, H. Manem, R. Karri, and R. Pino, "Leveraging memristive systems in the construction of digital logic circuits," *Proc. IEEE*, vol. 100, no. 6, pp. 2033–2049, Jun. 2012.

[18] C. Collier *et al.*, "Molecular-based electronically switchable tunnel junction devices," *J. Amer. Chem. Soc.*, vol. 123, pp. 632–641, Dec. 2001.

[19] G. Rose and M. Stan, "BA programmable majority logic array using molecular scale electronics," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 54, no. 11, pp. 2380–2390, Nov. 2007.

[20] E. Goto *et al.*, "Esaki diode high-speed logical circuits," *IRE Trans. Electron. Comput.*, vol. EC-9, pp. 25–29, Mar. 1960.

[21] T. Huisman, "Counters and multipliers with threshold logic," M.S. thesis, Delft Univ. Technol., Delft, The Netherlands, May 1995.

[22] J. Zhou, Y. Tang, J. Wu, X. Fang, X. Zhu, and D. Huang, "Design of counters based on memristors," in *Proc. Int. Conf. Inf. Syst. Eng. Manage. (ICISEM)*, 2013, pp. 483–486.

[23] L. Guckert and E. E. Swartzlander, "MAD Gates—Memristor logic design using driver circuitry," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, to be published.

[24] F. Corinto and A. Ascoli, "A boundary condition-based approach to the modeling of memristor nanostructures," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 59, no. 11, pp. 2713–2726, Nov. 2012.

[25] Y. Deng *et al.*, "RRAM crossbar array with cell selection device: A device and circuit interaction study," *IEEE Trans. Electron Devices*, vol. 60, no. 2, pp. 719–726, Feb. 2013.

[26] T. Devolder *et al.*, "Single-shottime-resolved measurements of nanosecond-scale spin-transfer induced switching: Stochastic versus deterministic aspects," *Phys. Rev. Lett.*, vol. 100, p. 057206, 2008.

[27] S. Kvatinsky, M. Ramadan, E. G. Friedman, and A. Kolodny, "VTEAM—A general model for voltage controlled memristors," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 62, no. 8, pp. 786–790, Aug. 2015.

[28] Z. Biolek, D. Biolek, and V. Biolkova, "SPICE model of memristor with nonlinear dopant drift," *Radioengineering*, vol. 18, no. 6, pp. 210–214, Jun. 2009.

[29] E. Lehtonen and M. Laiho, "CNN using memristors for neighborhood connections," in *Proc. 12th Int. Workshop Cellular Nanosc. Netw. Appl. (CNNA)*, Feb. 2010, pp. 1–4.

[30] M. Pickett *et al.*, "Switching dynamics in titanium dioxide memristive devices," *J. Appl. Phys.*, vol. 106, pp. 1–6, Oct. 2009.
[31] J. Yang, M. Pickett, X. Li, D. Ohlberg, D. Stewart, and R. Williams, "Memristive switching mechanism for metal/oxide/metal nanodevices," *Nature Nanotechnol.*, vol. 3, pp. 429–433, Jul. 2008.
[32] D. Strukov and R. Williams, "Exponential ionic drift: Fast switching and low volatility of thin-film memristors," *J. Appl. Phys. A*, vol. 94, pp. 515–519, Mar. 2009.

Lauren Guckert (M'08)received her B.S. degree from The University of Texas at Austin in Electrical and Computer Engineering in 20012 and a M.S. degree in Electrical and Computer Engineering with a concentration in Computer Architecture and Embedded Processors in 2015. She is currently continuing her Master's studies, working towards her Ph.D. degree. She also works as a Graduate Design Engineer at ARM INC in Austin, Texas.

Earl E. Swartzlander received the B.S. degree from Purdue University in 1967, the M.S. degree from the University of Colorado in 1969, and the Ph.D. degree from the University of Southern California in 1972, all in electrical engineering.

He is a professor of electrical and computer engineering at the University of Texas at Austin. In this position, he and his students conduct research in computer engineering with emphasis on application-specific processor design, including high-speed computer arithmetic, embedded processor architecture, VLSI technology, and nanotechnology. As of August 2015, he has supervised 44 Ph.D. students.

From 1975 to 1990, he held a variety of positions at TRW including the director of Independent Research and Development in the TRW Defense Systems Group, the manager of the Digital Processing Laboratory in the Electronics and Technology Division, and the manager of the Advanced Development Office in the System Development Division.

He was the editor-in-chief of the IEEE Transactions on Computers from 1990 to 1994 and was the founding editor-in-chief of the Journal of VLSI Signal Processing. In addition, he has served as an associate editor for the IEEE Transactions on Computers, the IEEE Transactions on Parallel and Distributed Systems, and the IEEE Journal of Solid-State Circuits.

He has been a member of the Board of Governors of the IEEE Computer Society (1987-1991), the IEEE Signal Processing Society (1992-1994), and the IEEE Solid-State Circuits Council/Society (1986-1991). He has been a member of the IEEE History Committee (1996-2004), the IEEE Fellows Committee (2000-2003), the IEEE James H. Mulligan, Jr., Education Medal Committee (2007-2011), the IEEE Awards Planning and Policy Committee (2011-2013), the IEEE Awards Board Awards Review Committee (2014 to present), and the IEEE Awards Board (2015 to present).

He has chaired a number of conferences. He is the author of two books, editor of 11 books and the author or coauthor of 84 refereed journal papers, 40 book chapters, and 303 conference papers. He is a life fellow of the IEEE. He has been honored with the IEEE Third Millennium Medal, the Distinguished Engineering Alumnus Award from the University of Colorado, the Outstanding Electrical Engineer and Distinguished Engineering Alumnus Awards from Purdue University, and the IEEE Computer Society Golden Core Award.