Dadda Multiplier Designs using Memristors

Lauren Guckert
School of Electrical and
Computer Engineering
University of Texas
Austin, Texas 78712
Email: lguckert@utexas.edu

Earl E. Swartzlander, Jr.
School of Electrical and
Computer Engineering
University of Texas
Austin, Texas 78712
Email: eswartzla@aol.com

***Abstract*—Memristors have recently become a leader in future system design due to their unique storage abilities and high density. This work presents an optimized Dadda Multiplier using memristors and IMPLY logic. The design reduces the baseline delay by 30% and complexity by 50% and has fewer than 60% the components of the CMOS design. The design is also pipelined to achieve 3x the throughput of CMOS.**

I. Introduction

Memristors were first hypothesized by Leon Chua [1] in 1971. Since then, research has focused on developing memristive devices with smaller form factor and faster switching times than current CMOS equivalents [2,3]. 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 [4-7]. However, recently, HP Labs showed that it is possible to implement all Boolean logic functions on memristors using the material implication, or IMPLY, operation [8].

Since this finding, an orthogonal path of research has begun to explore arithmetic applications using IMPLY and memristors [9-15]. However, all of the preliminary work that has been done has focused on individual logic gates and small circuits such as adders. In this work, an optimized memristor-based Dadda Multiplier design is proposed. Additionally, the implementation is modified to allow for pipelining, a key benefit of memristor-based logic approaches. The design is analyzed in terms of delay and complexity against the traditional CMOS design. Simulation results are also given to confirm functional correctness and provide a more concrete analysis of delay. The VTEAM model [14] is used for these simulations.

A. IMPLY memristor logic

The IMPLY operation is the most popular approach to memristor-based logic in modern literature. The IMPLY operation operates on two operands with a truth table as shown in Table I.

TABLE I: IMPLY Truth Table

P	Q	P IMP Q
0	0	1
0	1	1
1	0	0
1	1	1

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.

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

The IMPLY operation can successfully perform Boolean operations using the circuitry shown in Figure 1.

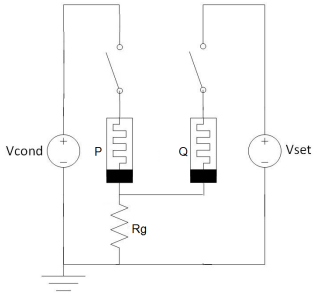

Fig. 1: Circuitry for the IMPLY operation P IMP Q

The circuitry requires two memristors, P and Q, and their drivers to hold the input operands and a pull-down resistor R_g. To store a binary value in a memristor, let a memristor with a high resistance represent a logical '0' value and a memristor with a low resistance represent a logical '1'. To execute the IMPLY operation, voltages are applied via the driver circuitry. The operation P IMP Q is performed by applying a voltage V_{cond} to memristor P and a voltage V_{set} to memristor Q concurrently where $V_{cond} < V_{set}$. A reset of a memristor is performed by applying V_{reset} and setting the memristor to high resistance, or '0'. For this work, let V_{cond} = 1.6V, V_{set} = 2.5V and V_{reset} = -5V. In an IMPLY operation, the result ends up in the Q memristor, overwriting the original input operand. By repeating this behavior, applying V_{cond} and V_{set} voltages

to two memristors at a time, all Boolean operations can be performed and more complex arithmetic units such as adders and multipliers can be constructed.

II. IMPLY Dadda Multiplier

No prior work exists for IMPLY-based Dadda multipliers so first a baseline design is implemented. The baseline design uses optimized full adders that have a delay of 21 steps and area of 7 memristors. Although numerous full adder implementations exist in literature, this implementation has the lowest complexity and delay [15]. The half adders consist of an XOR and an AND operation for a total of 11 steps and 11 memristors. The final stage addition uses the optimized ripple carry adder proposed in prior work [15] requiring 2N+19 steps and 7N+1 memristors. The initial AND operations are performed using standard IMPLY Boolean logic in 3 steps with 4 memristors. For a Dadda multiplication with S stages and an N-bit final addition, this results in a delay of 21S+2N+22. For an 8-bit multiplication, a Dadda multiplier has 64 AND gates, 4 stages comprising a total of 7 half adders and 35 full adders, and a final 14-bit ripple carry addition. The AND gates require 256 memristors, the half adders require 77, the full adders require 245, and the final ripple carry adder requires 99 memristors. Thus, the total delay is 134 IMPLY steps and the total complexity is 677 memristors.

The proposed design optimizes the baseline design to a delay of 14 IMPLY steps per stage rather than 21 steps for a total delay of 14S+2N+22. For an 8-bit multiplication, this results in a total of 106 steps, reduced from 134. Note that 47 of these steps are for the final addition, so the delay of the Dadda multiplier stages have been reduced by over 30%. This delay has been achieved by maximizing the parallelization of computations and minimizing the need for copy, reset, set, and sense stages during operation.

A. Design Optimizations

First, the individual adders leverage the fact that the inputs can be overwritten, unlike standard IMPLY-based addition. In the Dadda multiplier, the inputs to the adders are the results of AND operations or adders from previous stages. These inputs are used by a single adder instance so they can be overwritten without the risk of needing them for reuse later in the execution. Because of this, all operations to set and sense the outputs of stage S as inputs into the adders in stage S+1 are not necessary. This also eliminates the need for the memristors to hold these redundant values. Instead, the output memristors in stage S can be operated on directly during the IMPLY operations that use them in stage S+1. This effectively removes 2 memristors from each adder in the design.

Second, Boolean simplification is used to reduce the number of total IMPLY steps required to perform a half addition and full addition. The baseline required 14 steps and 19 memristors to perform two AND operations followed by a half addition and required 24 steps and 15 memristors to perform two AND operations followed by a full addition. The proposed design reduces this to 10 steps with 8 memristors and 15 steps with 13 memristors. Table III shows the execution steps for performing the AND operations followed by a half addition or full addition.

A and B represent the inputs to the AND gate and P, Q, and R represent the inputs to the adders. The first 3 steps execute the AND operation in parallel across all inputs. Thus after the first 3 steps, M0 and M1 hold the first input,P, and its inverse, NP, and M2 and M3 hold the second input Q and its inverse. For the full adder, M4 and M5 hold the third input R and its inverse. In a standard IMPLY-based ripple carry adder, the critical path lies along the carry propagation and thus the delay for the carry out bit is minimized. However, in the design of a Dadda multiplier, both the carry and sum out bits of the full adders are along the critical path. Thus, the execution steps are slightly different from the standard full adder to accomplish a mutually minimal delay for the two outputs. These adders are designed to produce not only the carry-out and sum bits but also their inverses in minimal delay. All four of these values are needed by subsequent stages in the multiplication. This changes the strategy for the order of Boolean operations to achieve the desired results. The inverse of the carry-out is produced as a byproduct of intermediate steps required for the carry-out bit in the half adders. Thus, only a single step is required at the end of the half adders normal execution to produce the inverse of the sum and provide all four outputs. The full adder requires two additional steps to produce the inverse of the carry-out and sum results for a total of 17 steps.

Note that the half adders are not on the critical path. Thus, the half adders favor memristor reuse instead of delay as a complexity optimization. The production of the inverses reduces the delay of each stage of the multiplication to 14 steps. In each stage after the first, the first three steps of the adders can be amortized. These steps are dedicated to resolving the partial products. For adders which require these as inputs, they can perform their AND operation a priori at the same time as the first stage. For adders which do not require these inputs and use intermediate values as operands instead, these three steps are not necessary as these values have been produced in the previous stage as described. Thus, the total delay is 17 steps for the first stage, 14 steps for each remaining stage, and 2N+19 steps for the final N-bit addition. Thus, the total delay is 14S+2N+22 steps. A waveform showing the resolution of the final product of an 8-bit Dadda multiplier for 0x65*0x43 = 0x1A6F is shown in Figure 2.

Numerous complexity optimizations were performed by reusing memristors within components and across components to reduce the baseline design's complexity of 677 memristors to 385 memristors. Recall that the half adders do not lie on the critical path of the multiplication stages and were optimized for memristor reuse instead of delay. These optimizations reduce the complexity of all 7 half adders by 5 memristors to only 8 memristors. Similarly, some adders in later stages in the pipeline use the result of one or more of the initial AND operands as inputs. All of these AND operations complete in the first 3 steps of execution, so the intermediate memristors used for these operations can be reused immediately and the

TABLE III: Execution steps for an IMPLY Dadda full adder and half adder for inputs A and B

STEP	HALF ADDER	GOAL	FULL ADDER	GOAL
1	B imp 0 ->M0		B imp 0 ->M0	
2	A imp (B imp 0) ->M0	A NAND B	A imp (B imp 0) ->M0	A NAND B
3	(A imp (B imp 0) imp 0) ->M1	A AND B results now live in P and Q	(A imp (B imp 0) imp 0) ->M1	A AND B results now live in P,Q,R
4	M0 imp M2 ->M2		M0 imp M2 ->M2	
5	M3 imp M0 ->M0	P NAND Q = NOT COUT	M3 imp M0 ->M0	P NAND Q
6	M1 imp M3 ->M3		M1 imp M3 ->M3	
7	Reset M1		M2 imp 0 ->M1	
8	M3 imp 0 ->M1		M3 imp M1 ->M1	P XOR Q
9	M2 imp M1 ->M1	P XOR Q = SUM	M1 imp M4 ->M4	
10	M0 imp 0 ->M4,M5	P AND Q = COUT	M4 imp 0 ->M6	
11	M1 imp 0 ->M6	P XNOR Q = NOT SUM	M0 imp M6->M6	COUT
12			M1 imp M5 ->M5	
13			M5 imp M1 ->M1	
14			M1 imp 0 ->M7	
15			M5 imp M7 ->M7	SUM
16			M6 imp 0 ->M8	NOT COUT
17			M7 imp 0 ->M9	NOT SUM

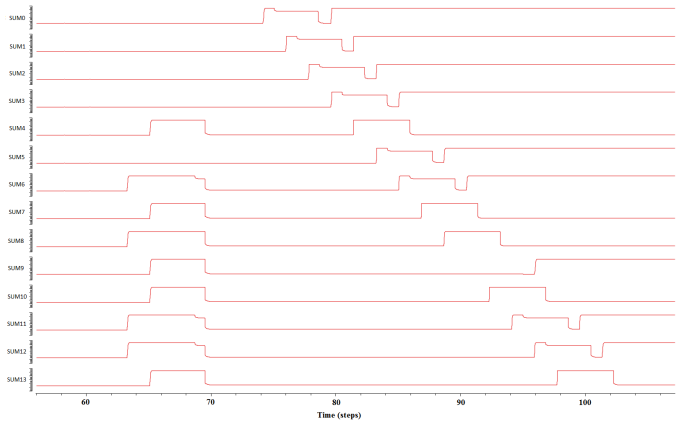

Fig. 2: 8-bit IMPLY Dadda multiplication execution for 0x65*0x43 = 0x1A6F

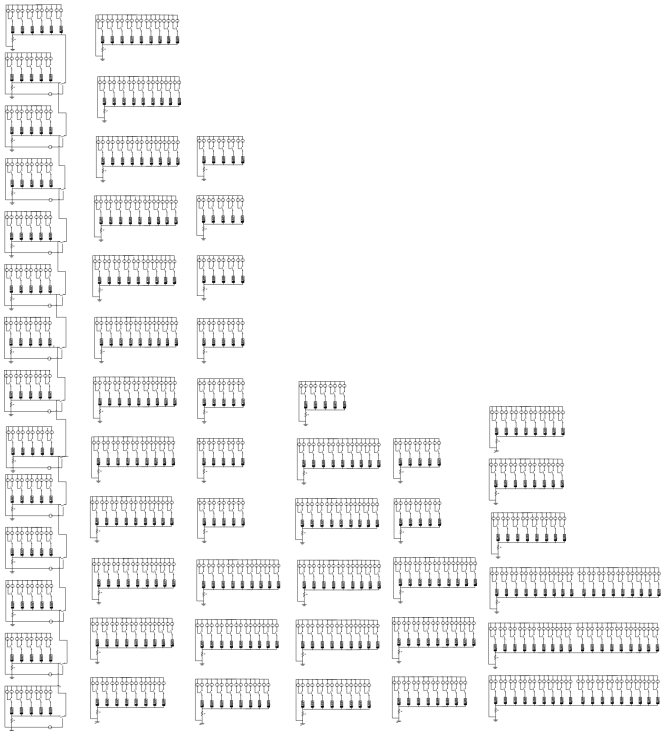

Fig. 3: 8-bit IMPLY-based Dadda multiplier schematic

output memristors can be reused once they are sensed. This saves 4 memristors per input in each of these full adders. As mentioned in the discussion of delay optimizations, some full adders use outputs from the previous stage as one or more of their inputs. These instances are optimized to remove unnecessary set and sense stages. Thus, these adders require as few as 5 memristors, one for each of the results the adder produces. In all, these optimizations reduce the overhead of the full adders in all the stages but the first to only 9 memristors or 5 memristors. In the first stage, the full adders require all 18 memristors since they are on the critical path. No reset operations or memristor reuse were performed on these adders in favor of reduced delay. Thus, the 3 full adders in the first stage use 18 memristors each, 9 full adders require 9 memristors, and the remaining 23 full adders require only 5 memristors. The total complexity of the full adders, half adders, and ripple carry adder is 385 memristors. A schematic for an 8-bit Dadda multiplier with the area optimizations is shown in Figure 3.

The stages traverse through the columns of the schematic with the initial stage on the right and the final 14-bit ripple carry adder on the left.

Although this design does not perform pipelining, pipelining is possible with IMPLY. In a memristor-based design, the intermediate values in the multiplication are stored in the memristors themselves rather than propagating through the design as voltages as is done in the CMOS design. Thus, without any changes to the circuitry or programming logic, the design can be altered to perform in a pipelined manner. Since the memristors are manipulated via drivers, each memristor can essentially be operated on disjointly, in parallel, as

needed. Thus, the Dadda multiplier can be pipelined at a stage granularity simply by changing when the drivers are applied to the adders. Now, each stage could begin a new addition as soon as it completes the previous addition and resets its memristors. Since the longest delay of any stage in the multiplier is 17 steps, a new multiplication can begin every 18 steps.

III. Analysis and Comparison

Table IV shows the complexity and delay comparison for the baseline and optimized 8-bit IMPLY Dadda Multiplier implementations as compared to CMOS.

TABLE IV: Complexity and delay for the 8-bit Dadda multiplier designs

	CMOS	IMPLY base	IMPLY opt
Steps	57	134	106
Complexity	2204 MOSFETs	677 memristors 682 drivers 783 switches	385 memristors 390 drivers 482 switches

Both the proposed IMPLY implementations require fewer components than the CMOS design, emphasizing the complexity benefits of memristor designs. The density and form-factor of memristors is generally smaller than CMOS transistors too, leading to a two-fold improvement in overall area. The optimized IMPLY design reduces the baseline design to nearly half of the original component count, over a 40% improvement as compared to the CMOS design.

It is important to note that a CMOS time step is not necessarily equivalent to an IMPLY time step. Thus, delay is reported in terms of step counts rather than total delay. The baseline IMPLY approach requires over twice the number of steps as compared to the CMOS design. This is due to the serialized nature of Boolean operations in the IMPLY context. The optimized implementation reduces this by 28 steps, or 30%, to offer a more competitive design. Recall that if pipelining is performed on the IMPLY design, the throughput of the design can effectively hide the latency of the multiplication such that a new multiplication can begin every 18 steps. In doing so, the IMPLY implementation requires less than 1/3 the number of steps as the CMOS design.

IV. Conclusion

Memristors are a new processing technology that have recently become popular as an alternative to traditional CMOS due to their favorable size and density. Although memory is the most common application for memristors, they are also able to perform logic using the IMPLY operation. This work presents an optimized implementation of a Dadda Multiplier using IMPLY memristor logic. The optimized design improves significantly over the baseline design, reducing the delay by 30% and the complexity by 50%, while offering a lower area alternative than the CMOS design. Also, the proposed design can be pipelined to offer greater throughput than CMOS.

Acknowledgment

The authors would like to thank the ECE department at University of Texas at Austin for the support.

References

[1] L. O. Chua, "Memristor-The Missing Circuit Element," *IEEE Transactions on Circuit Theory*, Vol. 18, pp. 507-519, 1971.

[2] F. Miao, J.P. Strachan, J.J. Yang, M. Zhang, I. Goldfarb, A. Torrezan, P. Eschbach, R. Kelley, G. Medeiros-Ribeiro, and R. Williams, "Anatomy of a Nanoscale Conduction Channel Reveals the Mechanism of a High-Performance Memristor," *Advanced Materials*, 2011.

[3] Y. Zhang, Y. Shen, X. Wang, and L. Cao, "A Novel Design for Memristor-Based Logic Switch and Crossbar Circuits," *IEEE Transactions on Circuits and Systems I: Regular Papers*, Vol. 62, pp.1402-1411, May 2015.

[4] Y. Zhang, Y. Shen, X. Wang, and Y. Guo, "A Novel Design for a Memristor-Based OR Gate," *IEEE Transactions on Circuits and Systems II: Express Briefs*, Vol. 62, pp. 781-785, Aug. 2015.

[5] X. Zhu, X. Yang, C. Wu, N. Xiao, J. Wu and X. Yi, "Performing Stateful Logic on Memristor Memory," *IEEE Transactions on Circuits and Systems II: Express Briefs*, Vol. 60, pp. 682-686, Oct. 2013.

[6] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "MAGIC Memristor Aided LoGIC," *IEEE Transactions on Circuits and Systems II: Express Briefs*, Vol. 61, pp. 1-5, 2014.

[7] Y. Yang, J. Mathew, S. Pontarelli, M. Ottavi and D. K. Pradhan, "Complementary Resistive Switch-Based Arithmetic Logic Implementations Using Material Implication," *IEEE Transactions on Nanotechnology*, Vol. 15, pp. 94-108, Jan. 2016.

[8] J. Borghetti, G. S. Snider, P. J. Kuekes, and J. J. Yang, "Memristive Switches Enable Stateful Logic Operations via Material Implication," *Nature*, Vol. 464, pp. 873-876, April 2010.

[9] K. Bickerstaff and E. E. Swartzlander, Jr., "Memristor-Based Arithmetic," *44th Asilomar Conference on Signals, Systems and Computers*, pp. 1173-1177, Nov. 2010.

[10] D. Mahajan, M. Musaddiq, and E. E. Swartzlander, Jr., "Memristor Based Adders," *48th Asilomar Conference on Signals, Systems and Computers*, pp. 1256-1260, Nov. 2014.

[11] A. Shaltoot and A. Madian, "Memristor Based Carry Lookahead Adder Architectures," *IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pp. 298-301, Aug. 2012.

[12] M. Teimoory, A. Amirsoleimani, J. Shamsi, A. Ahmadi, S. Alirezaee, and M. Ahmadi, "Optimized Implementation of Memristor-Based Full Adder by Material Implication Logic," *21st IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 562-565, Dec. 2014.

[13] M. Teimoory, A. Amirsoleimani, A. Ahmadi, S. Alirezaee, S. Salimpour, and M. Ahmadi, "Memristor-Based Linear Feedback Shift Register Based on Material Implication Logic," *Proceedings of 22nd European Conference on Circuit Theory and Design (ECCTD'2015)*, Aug. 2015.

[14] 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 Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 22, pp. 2054-2066, 2014.

[15] L. Guckert and E. E. Swartzlander, Jr., "Optimized Memristor-Based Ripple Carry Adders," *50th anniversary of the Asilomar Conference on Signals, Systems, and Computers* [Pre-Print].