The Dissertation Committee for Lauren Elise Guckert Certifies that this is the approved version of the following dissertation:

MEMRISTOR-BASED ARITHMETIC UNITS

Committee:

Earl Swartzlander Jr., Supervisor

Lizy John

Jack Lee

Michael Schulte

Nur Touba

MEMRISTOR-BASED ARITHMETIC UNITS

by

Lauren Elise Guckert, B.S.E.E.; M.S.E

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2016

Dedication

First, to my parents, Toni and Thomas Guckert, for their unwavering encouragement, understanding, and love throughout my life. Secondly, to my sister, who taught me to always strive for more but never forget to find a balance. To my friends and colleagues, for the laugh, memories, and reality checks. To Professor Swartzlander Jr., for his guidance, advice, and constant support throughout this whole journey. To Stafford, for getting me through the lowest lows and making a point to celebrate the highest highs. And last but not least, to my cat Winston, for always being there and never telling me I'm wrong. I wouldn't be here without any of you.

MEMRISTOR-BASED ARITHMETIC UNITS

Lauren Elise Guckert, Ph.D.

The University of Texas at Austin, 2016

Supervisor: Earl Swartzlander

The modern computer architecture community is continually pushing the limits of performance, speed, and efficiency. Recently, the ability to satisfy this endeavor with popular CMOS technology has proved difficult, and in many settings, impossible. The community has begun to explore alternatives to standard practices, researching new components such as nanoscale structures. Additional research has applied these new components and their characteristics to rethink the architecture of the latest technology, moving away from the Von Neumann architecture. A leading technology in this effort is the memristor.

Memristors are a new class of circuit elements that have the ability to change their resistance value while retaining knowledge of their current and past resistances. Their small form factor, high density, and fast switching times have sparked research in their applications in modern memory hierarchies. However, their utility in arithmetic has been minimally explored.

This dissertation describes the prior work in the exploration of memristor technology, fabrication, modeling, and application, followed by the completed research performed in the design and implementation of arithmetic units using memristors.

Implementations of popular adders, multipliers, and dividers in the context of memristors are designed using four approaches: IMPLY, hybrid-CMOS, threshold gates, and MAD gates. Each of these approaches has different tradeoffs and benefits for memristor-based design. Although the first three approaches have been defined in prior work, MAD gates are a novel application for memristors proposed that offer lower power, area, and delay as compared to prior approaches. This work explores these benefits for arithmetic unit design.

The details of each designs, simulation results, and analyses in terms of complexity and delay and power are presented. For arithmetic units which have been designed or presented in prior work, this research improves upon the design in each metric. Many of the designs are transformed and pipelined to leverage memristor characteristics and the various approaches rather than traditional CMOS and this is discussed in detail. Overall, the proposed designs offer significant improvements to traditional CMOS designs, motivating the effort to continue exploring memristors and their application to modern computer architecture design.

Table of Contents

List of Tables

List of Figures

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]. Orthogonally, research has explored various models and applications for these devices [4-9]. 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 [10-13]. However, recently, HP Labs showed that it is possible to implement all Boolean logic functions on memristors using the material implication, or IMPLY, operation [35].

Since this finding, an orthogonal path of research has begun to explore arithmetic applications for memristors [14-29]. Some of these works have focused on the IMPLY operation [14-19], while others propose alternative approaches that offer lower componential counts or step delay [20-29]. However, all of the preliminary work that has been done has focused on individual logic gates and small circuits such as adders.

Recently, some of these imply based implementations have come under scrutiny for their complex circuitry and multi-step operation [30, 31]. This has led to an exploration of other research paths to establish alternative memristor implementations for Boolean operations such as hybrid approaches [20-22], MAGIC gates [12] threshold gates [22, 23, 27], and others. None of these works provide thorough descriptions of complex designs, implementations, or optimizations for arithmetic units.

In this work, four distinct approaches to memristor-based arithmetic unit design are proposed and described that extend and optimize these prior works: IMPLY, hybrid-CMOS, threshold gates, and MAD gates. The first three approaches have been presented in prior work, but MAD gates are a novel contribution of this dissertation. In this work, MAD gates are introduced, analyzed, and compared against the alternative, pre-existing approaches to memristor logic. Then, the designs of numerous arithmetic units, including adders, multipliers, and dividers, are presented. Specifically, the adders studied are a ripple carry adder, a carry select adder, a conditional sum adder, and a carry lookahead adder. The multipliers studied are a shift-and-add multiplier, a Booth multiplier, an array multiplier, and a Dadda multiplier and the dividers studied are a binary non-restoring divider, an SRT divider, and a Goldschmidt divider. These designs were selected based on their proclivity to memristor optimizations as well as encapsulating a wide range of design tradeoffs. It is also shown how the MAD gates can be incorporated into a crossbar structure, simple to the IMPLY circuit.

For each arithmetic unit, optimized implementations are designed for the approaches described. In many cases, the traditional circuitry for the adder has been reworked to accommodate memristors and their properties. Additionally, some of the circuitry has been modified to allow for pipelining, a key benefit of some memristor-based logic approaches. These designs are analyzed in terms of delay, complexity, and power where applicable, and compared against each other and existing solutions. Simulation results are also given to confirm functional correctness and provide more concrete analyses of delay. For the implementations which have associated prior work, the proposed designs achieve better performance (delay and componential costs) than their prior counterparts. The remaining designs are the first to be discussed and proposed in literature. All designs are able to achieve comparable or improved delay over both traditional CMOS designs and prior work while decreasing the complexity required for the adders. For the IMPLY based approaches, a significant portion of the improvements are achieved by strategic operation ordering. For the hybrid CMOS approach, the majority of the improvements come from memristor reuse, whereas for the threshold gate approach, Boolean equation manipulation contributes the most. The MAD designs achieve the lowest delay and require the fewest components for every arithmetic unit, providing a significant contribution to memristor-based logic for modern systems. For the IMPLY and MAD approaches, pipelining also contributes heavily to the delay improvements when considering throughput.

The rest of the dissertation presents the prior work in this field, the proposed arithmetic designs and results, and suggestions for future work. Section 2 describes prior work on memristor-based arithmetic implementations via IMPLY operations, hybrid-CMOS gates, threshold gates, and others. Section 3 introduces MAD gates and provides motivation for their use as a lower complexity and lower delay approach than prior work. Section 4 describes the optimized designs for adders using the four aforementioned approaches and analyzes their step count and componential costs with respect to each other and CMOS. Simulation results and measurements are given. Sections 5 and 6 repeat this structure for the multipliers and dividers respectively. Section 7 discusses how MAD gates can be incorporated into a crossbar for logic-in-memory applications as well as proposes another alternative to memristor logic that can be performed in this context. Section 8 proposes future work to extend the findings in this dissertation to other components of modern systems and rethink the standard notion of computer architecture as a whole.

Simulation Methodology

For this research, popular tools and methodologies were employed to allow for reproducibility. The Cadence Virtuoso tool, version 6.1.6-64b, was used for all simulations. The most current version available at the time of the research was chosen. Virtuoso satisfied the basic requirements for a basic memristor simulation - a CMOS component library, the ability to measure power, delay, and component counts, use of external stimulation for the driver signals in the designs, and a memristor model.

The 45nm NCSU FreePDK suite was chosen for the CMOS component libraries due to its availability and wide use across prior literature. Numerous memristor models have been proposed and used, each with distinct properties. For this work the TEaM model [7] is used because it is capable of representing numerous models whereas other models focus on a single memristor fabrication. Also, the TEaM model is one of the few openly available models that offers a Verilog description for schematics. This is also the model used in the discussed prior works, allowing for direct comparisons between the works. One issue with the TEaM model is that it doesn't provide an accurate model of the power consumption. However, this is indicative of the infancy of this field rather than an issue with the model itself.

In order to use the TEaM model, various memristor parameters must be chosen carefully. Two such parameters are the resistances R_{high} and R_{low}, where $R_{high} >> R_{low}$ and a memristor with R_{high} resistance is considered in the 0 state and a memristor with R_{low} resistance is in the 1 state. The specific values of R_{low} and R_{high} depend on other properties of the memristors such as doping width and internal thresholds. A full list of the memristor parameters are given in Appendix A. The parameters given in the table are consistent for all designs discussed in this work and were chosen to match prior work.

All schematics were designed with the Virtuoso Schematic Editor L. All memristors and any other relevant values in the designs are initialized to 0, or high resistance at t=0ns. All of the driver signals (time and voltage) are generated using external files written in Microsoft Excel. All simulations were performed with transient analysis in the Virtuoso ADE L and the Spectre simulator. Specific values selected for voltages and other parameters are discussed for each design approach in the corresponding future sections.

When, implementing memristor-based designs, it is not practical to directly compare to the traditional CMOS design, and sometimes against other memristor designs. This is for multiple reasons. First, the concept of a "gate delay" does not exist in an IMPLY or MAD design since they are based on the applications of voltages to perform operations. Thus, the time delay of a step delay in these designs is not equivalent to the time delay of a gate delay in CMOS or other memristor-based designs. Second, memristors themselves are still largely not understood in terms of their interference and other practical limitations in a circuit. Thus, the actual dimensional area requirements of the memristor model are not available and make true area comparisons between designs impractical. Third, the components being used are often not the same and it is challenging to find a fair, impartial means for comparison. Lastly, the exact circuitry which should be included in the componential area counts for the designs are not always clear. For example, in the logic-in-memory context, there is disagreement on whether driver logic should be included and to what extent. Beyond driver logic, there is discussion on whether the logic to implement the driver logic itself should be measured. In this work the data path driver logic is reported and included in componential counts but the control logic for the drivers is not.

PRIOR WORK

There has been substantial prior work in CMOS arithmetic designs and in memristor technologies but very minimal research in the intersection of these two fields. The majority of the prior work in memristor-based logic has focused on the memory sphere, looking at ways to leverage the small form factor and high density of memristors to perform logic-in-memory. In most cases, the memristors are used in a crossbar structure where each memristor is used for a memory cell, uniquely identified by its row and column. Outside of the context of memory, there has been minimal research in memristor-based logic. The prior work on arithmetic units has focused on conceptual proofs and simple circuit examples, often times optimized or best-suited for use in memory. Examples of these are IMPLY, hybrid-CMOS, and threshold gates. Adding to the difficulty, memristor models are still in their infancy, with many different models existing for various applications and a lack of support for their use in simulation.

This section discusses prior work in memristor modeling for various applications and basic memristor gate design for the IMPLY, hybrid-CMOS, GOTO threshold, and MAD gate approaches. Analysis of the delay and complexity of each approach is given. The arithmetic units presented and implemented in this dissertation are presented at a high-level as background on their basic functionality. Where applicable, existing memristor-based implementations are described and analyzed.

Memristor Gate Implementations

The basic functionality of IMPLY, hybrid-CMOS, and threshold gate approaches to memristor-based designs are discussed in the following sections.

IMPLY Approach

The IMPLY operation is the most popular approach to memristor-based logic in modern literature. The IMPLY operation was first demonstrated to successfully perform Boolean operations using the circuitry shown in Figure 1 [13].

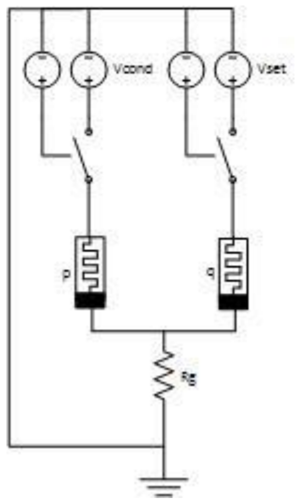


Figure 1: Circuitry for the IMPLY operation P IMP Q

The circuitry requires two memristors, P and Q, to hold the input operands. It is assumed that the value of P and Q have been set in the memristors a priori. These memristors are connected to a pull-down resistor, R_g. The value of R_g is selected based on the memristor parameters, driver voltages and other factors and the methodology is discussed thoroughly in prior work [19]. Memristors have a threshold T such that if the voltage drop across the memristor is greater than T, its internal resistance will change. If the voltage across the memristor is less than T, the internal resistance will remain unchanged. To successfully perform an IMPLY operation P IMP Q, the value of P must remain unchanged and the value of Q must change to hold the result. Thus, R_g must be selected such that the voltage drop across P is less than T and the voltage across Q is greater than T when the operation executed and R_g lies between R_{low} and R_{high}. For this work, R_g=2K ohms.

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 $V_{cond} < V_{set}$. For this work, let V_{cond} = 1.6V and V_{set} = 2.5V. The result ends up in the Q memristor, overwriting the original input operand. The corresponding truth table for the IMPLY operation is given in Table 1.

Table 1: Truth table for the IMPLY operation

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

Based on Table 1, the NOT operation can be computed by setting the input operand in memristor P and the value 0 in memristor Q and performing P IMP Q. Thus, as long as it is possible to clear a memristor to zero, the IMPLY operation becomes functionally complete. A memristor can be cleared by applying a V_{reset} voltage to the memristor where $|V_{reset}| > |V_{set}|$. For the purposes of this work, $|V_{reset}|$ = 5 V. Although it is possible to implement all Boolean operations using IMPLY, most take more than a single IMPLY step. The execution of Boolean operations with the IMPLY operation and their costs are given in Table 2.

Table 2: IMPLY implementations of Boolean operations [14]

Operation	Implementations	# Steps	# Memristors
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

Consider the OR operation. First, V_{cond} is applied to P and 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. 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. This is one key downside of the IMPLY operation.

To perform a COPY, two consecutive NOTs are performed. First, Q IMP 0 is performed onto a cleared memristor M_d such that M_d now holds (NOT Q). Then, M_d IMP 0 is performed onto another cleared memristor M_e such that M_e holds a copy of Q. Now the OR operation can be performed. In this way, rather than executing (P IMP 0) IMP Q and overwriting Q, (P IMP 0) IMP M_e is performed, overwriting its value but maintaining Q. Since this copy operation requires two memristors and 2 IMPLY steps, the total cost of an OR operation is 5 memristors and 4 steps. Similar reasoning can be applied to the remaining Boolean operations.

A major hindrance to both delay and latency is the high cost of the XOR operation in IMPLY because the XOR operation requires copying both operands before execution. An improved IMPLY implementation for the XOR operation [17] is given by

$$P \text{ XOR } Q = (P \text{ IMP } Q) \text{ IMP } ((P' \text{ IMP } Q') \text{ IMP } 0)$$

This reduces the latency to 7 and the memristor count to 6 assuming the initial operands cannot be overwritten.

HYBRID-CMOS APPROACH

One consequence of the IMPLY operation is its serialized nature. For example, in order to perform the OR operation it requires 4 steps instead of a single CMOS gate delay. Also, many memristors are used to hold intermediate results rather than useful data, another overhead of IMPLY. This is partially due to the fact that the IMPLY operation overwrites one of the input operands each step. Lastly, driver circuitry can become excessive depending on the scale of the circuits constructing for IMPLY.

In order to avoid the series of execution steps and accompanying driver circuitry for the IMPLY operation, prior work has proposed approaches that use both memristors and CMOS structures for logic operations in order to leverage the benefits of each domain. In hybrid-CMOS, memristors are used to construct AND and OR gates, while traditional CMOS is used for inverters (also called NOT gates) [20,21]. An AND and OR gate constructed from memristors are shown in Figure 2.

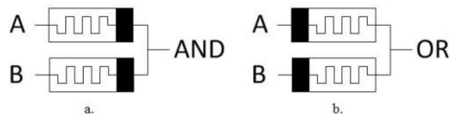


Figure 2: Hybrid-CMOS a) AND gate and b) OR gate

Voltages are applied to the A and B inputs to represent their values and a result voltage is output, where a low voltage is '0' and a high voltage is '1'. Let V_{low} = 0V and V_{high} = 3V. Note that the only difference between the two gates is the polarity of the memristors. The polarity of the memristors determines which direction of current flow is considered negative vs. positive, which in turn determines how the resistance of the memristors changes as the current changes.

Consider the AND gate. When A and B are both '0', or low voltage, no current flows through the memristors and the voltage at the AND output is a '0'. Similarly, when A and B are both '1', no current flows and the voltage at the output is a '1'. When A is '0' and B is '1', 3V flows from the B terminal to the A terminal. This causes the resistance in memristor B to increase and the resistance in memristor A to decrease. Thus the voltage at the AND output is low. The circuit behaves similarly when A is '1' and B is '0'. This is an AND operation.

For the OR gate, the only difference is the behavior of the memristor resistances in the case where one input is a 0 and one is a 1. In this case, the resistance in memristor A increases and the resistance in memristor B decreases, causing the opposite effect of the AND gate and the voltage at the output goes high. This is an OR operation.

Note that the operation of the gates relies heavily on the internal resistance of the memristors when the input voltages are applied. This is one concern with hybrid-CMOS designs. Designs based on this approach must take steps to ensure that the internal resistances of the memristors do not get too biased during execution. If this happens, the gates can begin to show high variability and misfunction. One way to avoid this issue is to periodically perform a “refresh” on the memristor gates, pulsing reset sequences to recalibrate the internal resistances.

Using the AND and OR gates presented and CMOS NOT gates, a hybrid-CMOS XOR gate can be constructed as shown in Figure 3.

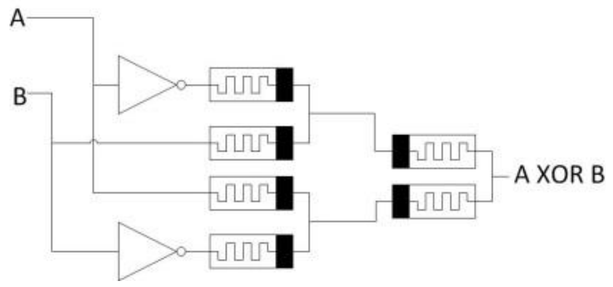


Figure 3: Hybrid-CMOS implementation of an XOR gate

This gate requires 4 MOSFETs and 6 memristors. One shortcoming of this gate design is the signal degradation as the operation propagates through the gates. For example, when A=’1’ (3 V) and B=’0’ (0 V) into an OR gate, the result at the output of the OR gate is not precisely 3V. When this result feeds into another gate as an input, it is no longer a full strength ‘1’. As the chain of gates or fanout increases in a design, the voltage strength can continue to decrease to a point where the gates can no longer differentiate it as a ‘0’ or ‘1’ properly. Depending on the specific parameters of the memristors as well as the previous and current propagated values (as described above), the signals degrade at different rates. Thus, a long series of hybrid-CMOS gates cannot be concatenated without a CMOS NOT gate or CMOS buffer to restore the signal. This increases both delay and complexity. Secondly, vias are required to transition from the CMOS domain to the nanoscale memristor domain and back.

THRESHOLD GATE APPROACH

Threshold gates are studied because they lend themselves to the operations in adder computations by simplifying logic and thus reducing gate delays. They also do not exhibit the serialization issues of the IMPLY operation or the signal degradation or CMOS issues of the hybrid-CMOS approach. However, threshold gates are still in their infancy stages of fabrication and are discussed here as an emerging technology in memristor-based logic design.

A threshold gate with threshold T is a gate that behaves as follows: For N inputs, X_0 to X_N, with weights w_0 to w_N,

$$\text{Output} = (\sum_{i=1}^{n} w_i * X_i) > T \qquad (1)$$

Two implementations of threshold gates are explored: GOTO pairs and CSTG gates.

Threshold Gates Implemented with GOTO pairs

Memristor-based threshold gates can be constructed from a structure known as a GOTO pair [26]. GOTO pairs are a convenient option for memristor-based gates because they require lesser area and removing the need for internal vias. One shortcoming is that they are currently difficult for metal-oxide-based fabrication. A deeper explanation of GOTOs can be found in prior work [26] but an example of a threshold gate is shown in Figure 4.

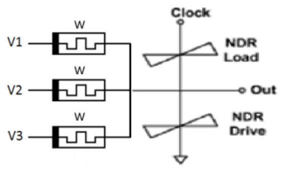


Figure 4: Majority gate implemented with memristors and a GOTO pair

The gate requires one memristor for each input, V1, V2, and V3, each with a memristive weight Wi. The weight for each memristor is selected based on their relative weights in the desired Boolean equation. For example, for a simple majority gate, each weight Wi would be equal to 1.

NDR Load and NDR Drive are two resonant tunneling diodes which perform the detection. These diodes create two stable operating points which essentially serve as logical '0' and '1' for the output of the threshold gate. If the sum of the incoming currents, SUM(Vi*Wi) is above the threshold, the output will pull toward the the logical '1' point, else it will pull toward the logical '0' point.

NOT logic is not implemented with this scheme. Although preliminary research has shown that it is possible to perform a NOT with memristors and GOTO pairs, it is highly sensitive and exhibits high variability. Thus, for this work, NOT logic is performed with traditional CMOS logic in the context of threshold designs.

Threshold Gates implemented with CSTG gates

Due to the difficulties of GOTO pair fabrication, a CMOS buffer is considered for the threshold detector. This can be implemented using Charge Sharing Threshold Gates (CSTGs) as shown in prior work [23] such as the one in Figure 5.

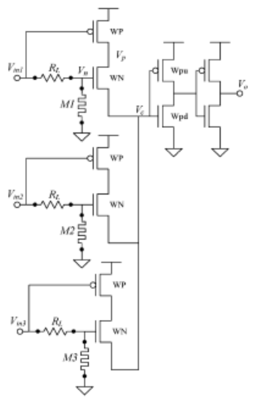


Figure 5: Example CSTG for threshold detection

A single CSTG requires 4 MOSFETs for the output plus 1 memristor, 1 resistor, and 2 MOSFETs per input. An N-input threshold gate requires N memristors, N resistors, and 2N+4 MOSFETs. Although greater in complexity, CSTGs generate both the output and its inverse. This can be leveraged to achieve lower area and lower delay in some arithmetic designs as described in later sections.

ALTERNATIVE APPROACHES

Two other alternative memristor-based gate implementations are MAGIC gates and Zhang et al.'s designs [3, 10, 12]. However, these gates are intended for use within memory and their correctness is based on this assumption. The functionality of the circuits relies on the independence of the gate to control the current and threshold behavior in conjunction with the polarity of the memristors. For circuits with a series of concatenated gates and operations, it is unclear how these types of memristor implementations can be used. The A and B operands cannot be used as inputs to multiple gates concurrently, disallowing for fanout into parallel operations. If the inputs were used to perform multiple Boolean operations in parallel, the behavior of each gate would affect the others and the results could be incorrect. Also, there is no given implementation for an XOR or XNOR gate. For these reasons, MAGIC gates and similar approaches are not explored as viable options for arithmetic unit design.

Another work proposed CRS cells as an alternative to memristors in the context of a crossbar [13]. However, each CRS cell uses two memristors and requires initialization on all operands before Boolean computation. Also, the result is stored in one of the operands. The underlying operation is only able to implement NAND-AND and NOR-OR gates, each of which requires 3 steps. All other Boolean operations must be constructed from a series of these operations, requiring a total of N+2 steps where N is the number of NAND and NOR operations required for the Boolean operation. The devices also have 4 different threshold voltages, making voltage detection and sensitivity more difficult and adding more voltages to the architecture. Also, their proposed designs for CRS-based adders and multipliers fail to improve over the designs proposed in this work. For these reasons, CRS cells are not considered in this work.

Ripple Carry Adders

Ripple carry adders are considered the baseline adder model and prioritize simplicity over performance. An example of a 4-bit ripple carry adder can be seen in Figure 6 [14].

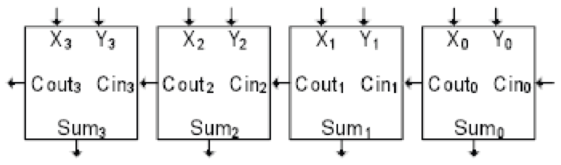


Figure 6: Ripple carry adder diagram

Each bit has a full adder with a sum and carry-out output. The carry-out output of each bit j is used as the carry-in input for bit j+1. The following sections discuss a traditional CMOS implementation of a ripple carry adder followed by the three proposed memristor-based approaches.

A traditional N-bit CMOS ripple carry adder requires 9 gates for each full adder as shown in Figure 7. This adder has a delay of 2N+4 for the final sum and requires 9N gates or equivalently 34N MOSFETs.

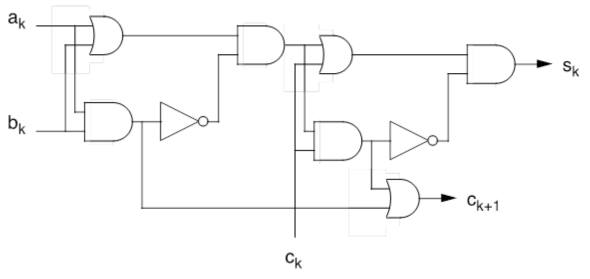


Figure 7: Traditional CMOS implementation of a full adder

Memristor-based ripple carry adders can be built from the gates discussed in Section 2.2.

PRIOR IMPLY RIPPLE CARRY ADDERS

The majority of the work in designing and optimizing IMPLY-based ripple carry adders has focused on minimizing memristors rather than delay. These approaches serialize the IMPLY steps of each bit in order to reuse memristors across steps and minimize the number of overall memristors required. This naturally results in an increased runtime in comparison to a parallelized approach. The state-of-the-art implementation [17] requires only 27 memristors for an 8-bit adder, but it requires 184 IMPLY steps. This delay is significantly higher than the traditional CMOS implementation's delay of 20 steps.

Minimal prior work has focused on parallel implementations of IMPLY ripple carry adders. Such an implementation was originally shown for an 8-bit RCA by [15], taking 35N memristors and 8N+12 steps. This design was later improved to require 9N memristors and a delay of 5N+18 IMPLY steps [19]. One consequence of parallel implementations is increased driver complexity in order to maintain independence among the bits' parallel computations.

PRIOR HYBRID-CMOS RIPPLE CARRY ADDERS

Minimal prior work exists in the design of hybrid-CMOS ripple carry adders due to the signal degradation issues inherent to hybrid-CMOS gates. The two works that have explored such designs require 16N memristors, 18N MOSFETs and 10N vias for an N-bit adder with the same gate delay as the traditional CMOS implementation. An example full adder circuit from these works is shown in Figure 8 [21].

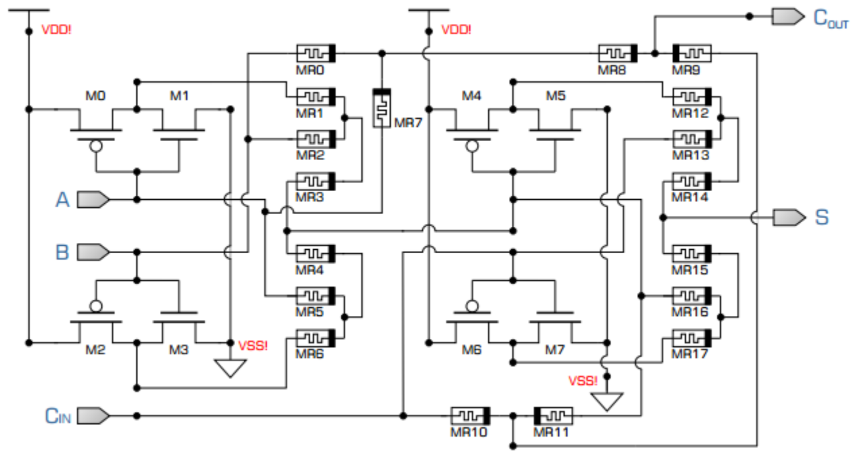


Figure 8: Hybrid-CMOS implementation of a full adder

PRIOR THRESHOLD GATE RIPPLE CARRY ADDERS

No known prior work has proposed a ripple carry adder implemented with memristor-based threshold gates.

Carry Select Adders

The carry select adder divides the inputs into blocks and forms two sums and two carry outs for each block in parallel (one block with a carry in of 0 and the other block with a carry in of 1). For the block that has a carry in of 0, the carry out bit is equivalent to the Generate bit, or G, as familiarized in carry lookahead adders. For the block that has a carry in of 1, the carry out bit is equivalent to the Propagate bit, or P. The carry out from the previous block controls a multiplexer that selects the appropriate sum. The carry out from the previous block is then propagated to the following block by the equation G+PC. Figure 9 shows an example for a 16-bit carry select adder with 4-bit block sizes.

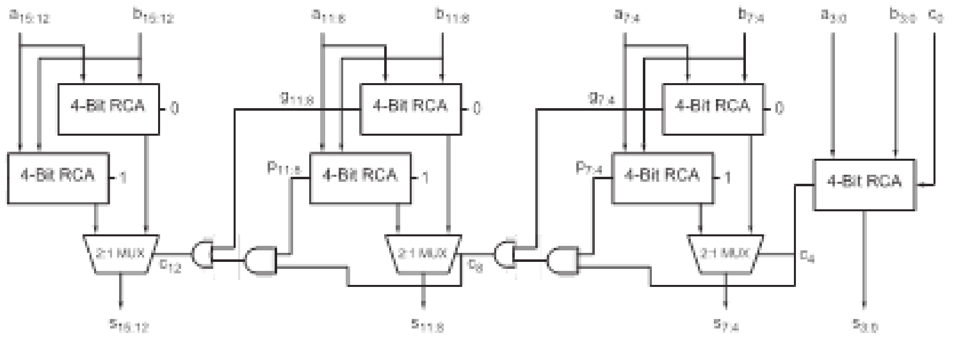


Figure 9: 16-bit carry select adder

The traditional CMOS implementation of an N-bit carry select adder with block size k requires 21N-12k+3N/k-5 gates with a delay of 2k+2N/k+2. The design is optimized for delay when the block size k is size √N resulting in 21N-9√N -5 gates and delay of 4√N+2. There is no known prior work on designing memristor-based carry select adders.

Conditional Sum Adders

Conditional sum adders leverage hierarchical multiplexers in order to improve the carry propagation time over ripple carry adders and other adder designs. The first bit uses a full adder cell since the carry in bit is known at time 0. The remaining sets of input bits each feed modified half adders (MHAs) which produce 4 output bits, C_0, C_1, S_0, and S_1. A modified half adder is shown in Figure 10 [15].

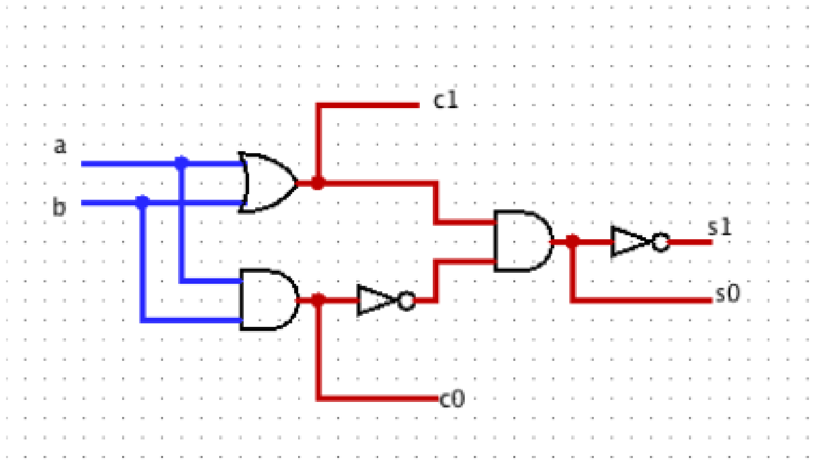


Figure 10: Modified half adder circuit

C_0 and S_0 correspond to the carry out and sum out bits if the carry in bit is a 0. C_1 and S_1 correspond to the carry out and sum out bits if the carry in bit is a 1. Each of these four output bits can be generated as soon as inputs A_i and B_i arrive at time 0. These outputs then feed multiplexers that are selected by the previous carry in bits. This minimizes the carry propagation delay to the delay of a multiplexer. An example of an 8-bit conditional sum adder is given in Figure 11.

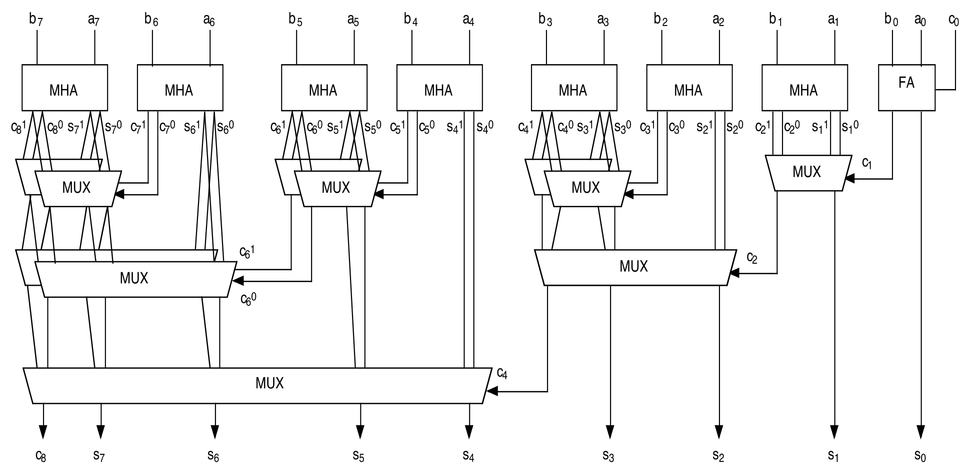


Figure 11 – 8-bit conditional sum adder

The delay of an N-bit conditional sum adder is proportional to $\log_2(N)$, an improvement over ripple carry adders and other designs. This comes at the expense of great complexity for the hierarchy of multiplexers.

A naive IMPLY-based conditional sum adder implementation was proposed [11] that required 454 memristors and a delay of 39 for an 8-bit adder and 975 memristors with a delay of 46 for a 16-bit adder. This implementation was not verified or optimized. No prior work in hybrid-CMOS or threshold gate approaches for memristor-based conditional sum adders is known.

Carry Lookahead Adders

Carry lookahead adders were first explored as an alternative to ripple-carry adders to eliminate the long delay imposed by the carry chain [32]. However, carry lookahead adders require significantly more complexity than ripple carry adders. Given the smaller form factor of memristors, a memristor-based carry lookahead adder could offer the improved performance without the area consequences. A diagram for a carry lookahead adder is shown in Appendix B.

Carry lookahead adders accelerate delay from O(N) to O(logN) by defining Propagate, P_i, and Generate Bits, G_i, and combining these bits into larger blocks. For each bit i in the addition, P_i and G_i have the following equations

$$P_i = A_i \text{ XOR } B_i \qquad (2)$$

$$G_i = A_i \text{ AND } B_i \qquad (3)$$

These bits can be produced in a single CMOS gate delay. Then, each intermediate carry out bit can be represented in terms of these P_i and G_i bits coupled with C_0, the initial carry in bit. This approach can be extended in logarithmic fashion to compute every carry out bit by defining Group Propagate, GP_i, and Group Generate bits, GG_i. These bits are similar to their individual counterparts except they represent propagate and generate bits for an adder block rather than a single bit. The corresponding equations for GP and GG are

$$GP=P_0*P_1*P_2*P_3 \qquad (4)$$

$$GG = G_3 + G_2*P_3 + G_1*P_2*P_3 + G_0*P_1*P_2*P_3 \qquad (5)$$

Prior work implemented an IMPLY based carry lookahead adder that reduced the delay for a 4-bit adder to 25 for the carry-out and 32 for the sum [16]. Area and complexity numbers were not given for this implementation. There are no known prior works for memristor-based carry lookahead adders implemented with hybrid-CMOS gates or threshold gates.

Memristor-Based Multipliers

Minimal to no prior work exists for memristor-based designs for multipliers. The complexity of these designs coupled with the complexity of memristor models and their programming leads to a high design overhead. However, there have been a few prior works that have explored multipliers in the context of memristors for the IMPLY approach.

One work [33] leverages the analog nature of memristors to store decimal values as resistance into memristors using op-amp circuitry. By varying the circuitry, the addition, subtraction, multiplication, and division operations can be achieved. However, this approaches requires reliability, consistency, and highly-sensitive components. The approach also requires extensive conversion circuitry and CMOS for the op-amps and other components, increasing the area. This dissertation uses memristors for both logic and state and attempts to remove all necessary CMOS when possible. The designs presented in this work also do not treat the memristor as an analog storage device due to the impracticality of fine-grained conversion, programming, and detection circuitry.

SHIFT AND ADD MULTIPLIERS

Shift-and-add multipliers consist of three main components, an adder (usually a ripple carry adder), a multiplexer, and shift registers. Both the product and the multiplier reside in a shift register. The multiplicand resides in a standard register. An example diagram for a shift-and-add multiplier is shown in Figure 12.

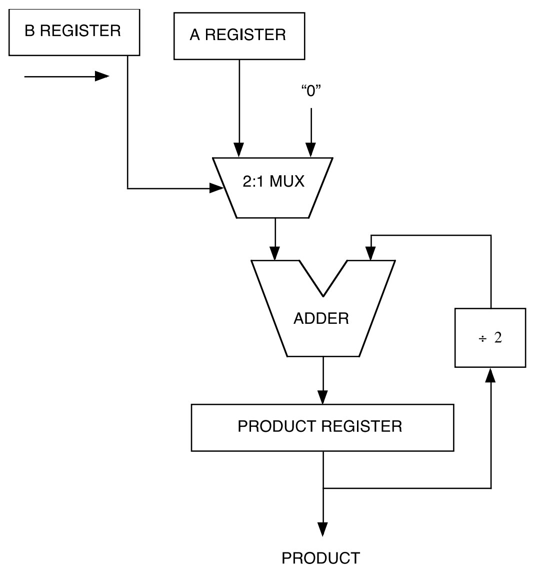


Figure 12: shift-and-add multiplier

At each clock cycle, the product and multiplier (in the B register) are shifted one bit to the right. The bit shifted out of the multiplier register is used by a mux to select either the multiplicand or the value 0 to be added to the current product register. Then, the addition is performed and the result stored into the product register. An N-bit multiplication is performed by repeating this N times. Booth multiplier is a slight variation on this multiplier for increased radices. Booth multiplier performs in the same manner but shifts multiple bits and resolves multiple bits per iteration. No known prior works have explored memristor-based shift-and-add multipliers or Booth multipliers.

ARRAY MULTIPLIERS

Array multipliers are multipliers which use an array of half adders and full adders to accomplish multiplication. Array multipliers extend an NxN multiplication operation into N additions. This is similar to the functionality of a shift-and-add multiplier except that the additions are pipelined in order to reduce the delay. In order to achieve this pipelining, the adder circuit is replicated N times, creating an array of (N-1)xN adders. A diagram for a 4x4 array multiplier is given in Figure 13.

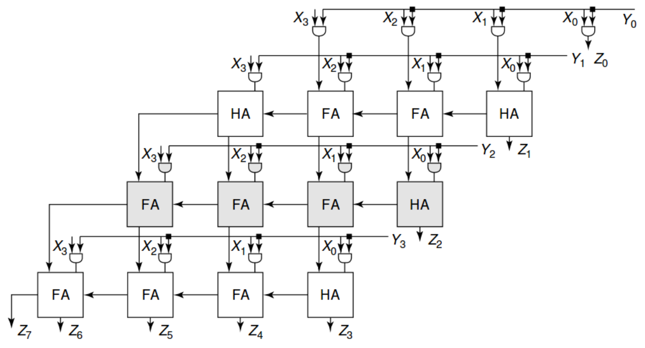


Figure 13: 4-bit array multiplier

One prior work [14] designed a 4-bit by 4-bit IMPLY-based array multiplier which requires 73 steps to complete the multiplication. This work does not confirm the design with simulation nor does it provide analysis on area or power. Lastly, the design does not perform any optimizations or extend the design to larger multipliers, rendering it inapplicable for most practical, modern architectures.

DADDA MULTIPLIERS

Dadda multiplier is a popular multiplier for fast computations that uses stages of addition to perform the overall multiplication. Each stage consists of full adders and half adders, each performing independent additions based on inputs from the previous stage. It is similar in principle to Wallace Multipliers, but places a greater emphasis on decreasing area. It accomplishes this by limiting the reduction and thus number of adders in each stage of the multiplication. Once the reduction has completed and only two rows of operands remains, a standard adder is used to complete the multiplication.

An example of a Dadda multiplier reduction for an 8-bit multiplication is given in Figure 14.

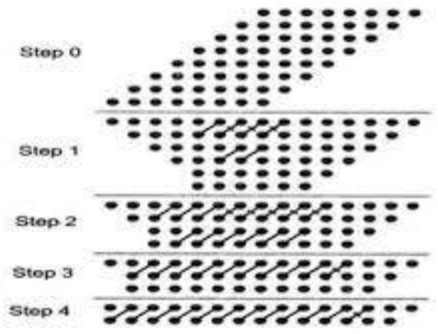


Figure 14: Dadda multiplier execution for an 8-bit multiplication

Each dot in step 0 represents a partial product A_iB_i from the original inputs A and B. A diagonal line represents a full adder and a diagonal line with a cross-stitch represents a half adder. You can think of a full adder as “swallowing” three input dots and outputs two output dots, one sum dot In its own column, and one carry out dot in the successive column. A half adder performs the same except it only “swallows” two input dots. In the figure above, after step 4, an adder performs the final 14 bit addition to resolve the product.

No known prior work exists in the exploration of memristor-based Dadda multipliers.

Memristor-Based Dividers

Similar to memristor-based multipliers, nearly no prior work exists in the design of memristor-based dividers. There is no known prior work at this time for memristor-based dividers with the exception of one work [33]. Similar to this work's multiplier design, the divider design treats the memristor as an analog storage device and requires extensive conversion circuitry and CMOS to program and read the value stored.

This dissertation takes an orthogonal approach to memristor-based dividers, using memristors for both logic as well as storage and eliminating CMOS when possible.

Binary Non-Restoring Dividers

Binary non-restoring dividers are dividers which execute serialized subtractions to resolve the quotient and remainder for a given dividend and divisor one bit at a time. The difference between a binary non-restoring divider and a binary restoring divider is that the non-restoring design does not subtract the divisor from the dividend unless the result remains positive. A diagram for a binary non-restoring divider with a 2n bit dividend and an n bit divisor is shown in Figure 15.

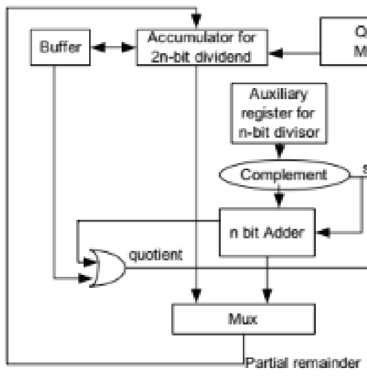


Figure 15: Binary non-restoring divider for a 2n-bit dividend and n-bit divisor

The design consists of an n-bit ripple carry adder, a 2n-bit shift register, and an n-bit multiplexer. The ripple carry adder first performs a 2's complement of the divisor to enable subtraction. Next, the adder adds this value to the most significant n bits of the dividend. If the most significant bit of the sum is 0, implying a positive result, the dividend is updated with the sum and the quotient bit is set to 1, else it is not. Then, the shift register is shifted to the left and the quotient bit is shifted in.

In general, binary non-restoring dividers offer lower area than alternative divider designs at the expense of a slower performance. Thus, for the designs discussed in this work, a greater emphasis is placed on area optimizations than delay optimizations. There is no known prior work in the design of a memristor-based binary non-restoring divider.

SRT DIVIDERS

SRT division is similar to non-restoring division but reduces the number of iterations by increasing the radix of operation. The quotient is represented in binary signed digit form. For this work radix-2 SRT division is considered. First, the 2's complement of the divisor is computed. Next, the value V = 2*(dividend) is computed in order to resolve a quotient bit Qi and the running sum. If $V > 0.5$, $Qi = 1$, if $V < -0.5$, $Qi=-1$, else $Qi = 0$. If $V > 0.5$, the 2's complement of the divisor is added to V, if $V < -0.5$, the divisor is added to V, else 0 is added. To perform the comparison, a 4-to-1 mux is used to select either the divisor, the 2's complement of the divisor, or 0 to be added to V. The select lines of the mux are the 2 most significant bits of V. If the select lines are 00 or 11, the value 0 is selected. If the select lines are 01, the 2's complement of the divisor is selected, and if the select lines are 10, the divisor is selected.

The next iteration begins by computing V_{i+1} as twice the current value of V_i. When the division completes, the quotient is converted to binary. Last, the remainder is checked for sign. If the remainder is negative, the divisor is added back to the running sum and 1 is subtracted from the quotient to undo the effects of the final subtraction. A flowchart for this is shown in Figure 16.

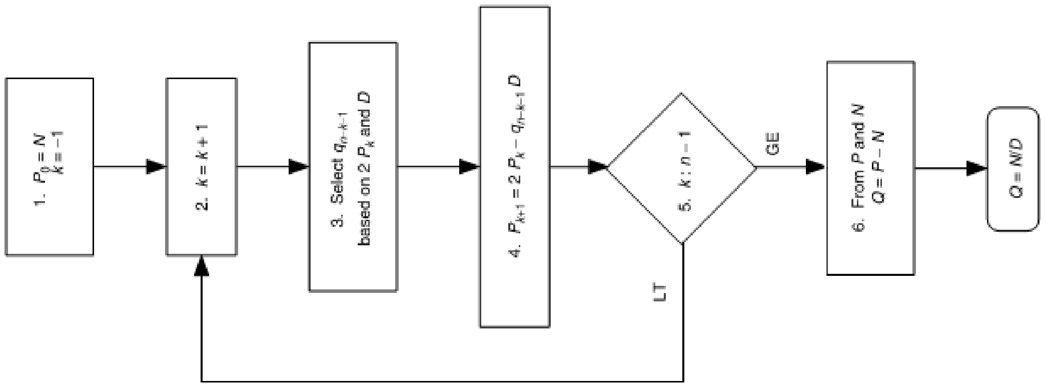

1. $P_0 = N$ $k = -1$
2. $k = k + 1$
3. Select q_{n-k-1} based on 2 P_k and D
4. $P_{k+1} = 2\, P_k - q_{n-k-1}\, D$
5. $k : n - 1$
GE
LT
6. From P and N $Q = P - N$
$Q = N/D$

Figure 16: Flowchart for a radix-2 SRT divider

In this work, on-the-fly conversion [34] is used to compute the binary quotient during the execution of the iterations rather than performing the conversion at the end of the division. This effectively hides the latency of the final quotient conversion. On-the-fly conversion retains information on the results of previous iterations to correctly convert the running binary signed digit quotient as the digits of the quotient are resolved in each iteration. An example of on-the-fly conversion is shown in Table 3.

Table 3: On-the-fly conversion of BSD to binary for SRT division

j	q_j	Q[j]	QM[j]
0		0	0
1	1	0.1	0.0
2	1	0.11	0.10
3	0	0.110	0.101
4	1	0.1101	0.1100
5	-1	0.11001	0.11000

J is the iteration, Qj is the quotient result, Q[j] is an intermediate value, and QM[j] is the final quotient in binary form. In this approach, the final remainder check is still necessary.

The equations which corresponding to Q[j] and QM[j] are as follows:

$$Q[j+1] = \begin{cases} (Q[j], q_{j+1}) & \text{if } q_{j+1} \geq 0 \\ (QM[j], (r - |q_{j+1}|)) & \text{if } q_{j+1} < 0 \end{cases} \quad \textbf{(6)}$$

$$QM[j+1] = \begin{cases} (Q[j], q_{j+1} - 1) & \text{if } q_{j+1} > 0 \\ (QM[j], ((r-1) - |q_{j+1}|)) & \text{if } q_{j+1} \leq 0 \end{cases} \quad \textbf{(7)}$$

Where r is the radix of the operation, in this case 2. All of the presented work on SRT dividers in this dissertation uses the on-the-fly conversion algorithm to reduce the delay and complexity of the designs. There is no known prior work in the design of a memristor-based SRT divider.

GOLDSCHMIDT DIVIDERS

Goldschmidt division is a fast division technique that resolves twice as many bits per iteration, significantly reducing execution time for large sizes. Goldschmidt uses repetitive multiplication of the normalized dividend N and divisor D by a factor F until the divisor converges to 1, with the dividend converging to the quotient. In each iteration, the value F is updated using the equation F_i=2-D_i. A flowchart can be seen in Figure 17.

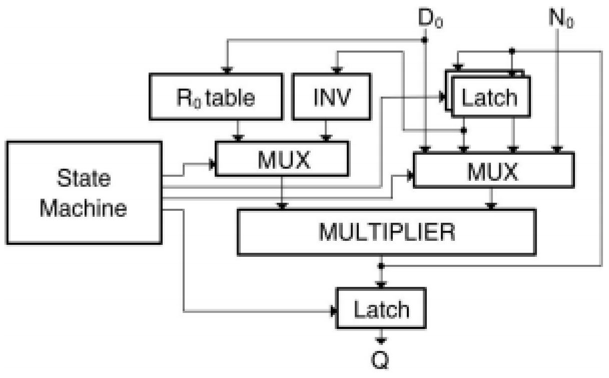


Figure 17: Goldschmidt division flowchart

In total, a division that executes X iterations requires 1 shift, 2X multiplications, and X subtractions. For this work it is assumed that both the dividend and divisor have been normalized a priori. This work focuses on 8-bit Goldschmidt dividers and uses Dadda multipliers for the fast multipliers. There is no known prior work in the design of a memristor-based Goldschmidt divider.

MAD GATES[1]

This dissertation introduces MAD gates, or Memristors-As-Drivers gates, to achieve lower power, complexity, and delay than previous memristor-based gate designs. MAD gates combine the benefits of the IMPLY operation with standard driver circuitry to decrease delay. MAD gates, or Memristors-As-Drivers gates, were introduced to overcome the long delays of the IMPLY operations and the fanout, signal degradation, and buffering issues in hybrid-CMOS operations.

[1] L. Guckert and E. E. Swartzlander, Jr., "MAD Gates - Memristor Logic Design Using Driver Circuitry," *IEEE Transactions on Circuits and Systems II: Express Briefs*, Vol. 99, pp.1, Apr. 2016. This section is an extension of previously published work for which Lauren Guckert is the principal author and responsible for all content.

Gate Structure

A MAD gate implementation of the AND operation is given in Figure 18.

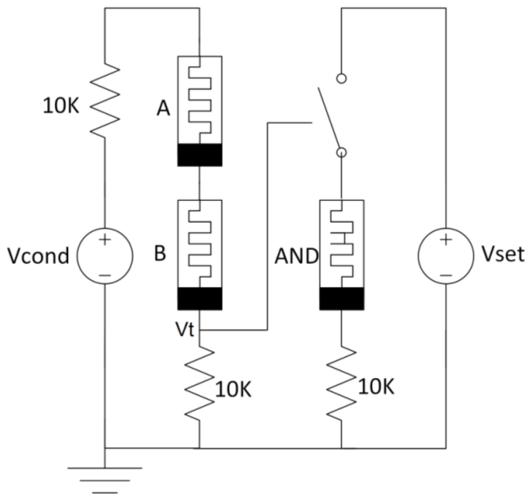


Figure 18: MAD AND gate

The two input operands are stored in memristors A and B and a switch paired with a third memristor, AND, holds the result. Assume that the values of A and B are preloaded into the input memristors. This can be done with standard IMPLY set or copy operations.

Similar to IMPLY, a pull-down resistor R_g is used but is now placed on each memristor. In this example, R_g=10K ohms. Also, V_{cond} and V_{set} are used as voltage drivers in the circuit and are selected by the same methods so V_{cond} = 1.6V and V_{set} = 2.5V. However, these drivers are used in a different way than in the IMPLY operation.

To perform the AND operation, the read voltage V_{cond} is applied to the input memristors in series. At the same time, a voltage V_{set}, is gated to the n terminal of the result memristor. The V_{set} voltage on the output memristor is gated by the voltage of the input circuit at node V_t. If the voltage sensed at node V_t is greater than the threshold of the switch on the output memristor, the switch will close and the V_{set} voltage will set the result memristor to a logical '1', else the memristor will remain a logical '0'. Thus the threshold of the switch, called V_{apply}, must be set accordingly. In this circuit, when both input memristors are logical '1', the voltage at node V_t is 16/22 V. If both inputs are logical '0', the voltage is 16/220V and if one of the memristors is '1' and the other is '0', the voltage of node V_t is 16/121V. Thus, to perform the AND operation, V_{apply} must be chosen such that $16/121V < V_{apply} < 16/22V$. Here, V_{apply} is chosen to be 0.5V.

The same circuit can be modified to accomplish the remaining Boolean operations by varying the value of V_{apply}. For example, for the OR operation, V_{apply} should be selected such that $16/220 < V_{apply} < 16/121$ V. This way, if either or both of the input memristors are a '1', the voltage at V_t will be above the threshold, and the result memristor will be set to '1'.

The circuit can also be used to accomplish a COPY operation. The NOT and COPY operations only require a single input memristor, but take the same form otherwise. The respective circuits for the OR, XOR, NOT, and COPY operations are shown in Figure 19.

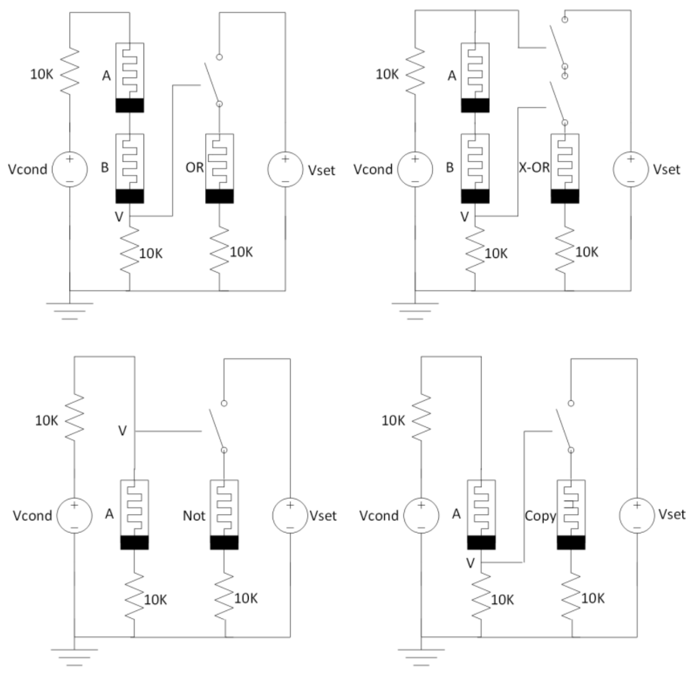


Figure 19: MAD OR, XOR, NOT, and copy gates

This design offers a uniform standardized cell for the inputs that can be configured with a threshold that depends on the gate and application. In addition to uniformity, these circuits offer improved latency over prior approaches: IMPLY operations, hybrid-CMOS, Zhang et al., and threshold gate implementations. All Boolean operations can be performed in 1 IMPLY delay, including the XOR operation. Although MAGIC gates also offer a single step delay, operations on the same inputs cannot be performed in parallel and the XOR operation is not implemented. A full delay comparison of the gate delays are given in Table 4.

Table 4: Delay comparisons for memristor-based Boolean gates

Operation	IMPLY	Hybrid-CMOS	MAGIC	Zhang	Threshold	MAD
NAND	2	2	1	3	2	1
AND	3	1	1	1	1	1
NOR	5	2	1	3	2	1
OR	4	1	1	1	1	1
XOR	8	3	N/A	N/A	3	1
NOT	1	1	1	2	1	1

The latencies are given in terms of the number of steps to complete the operation, which are not necessarily equivalent across the various approaches. For example, a hybrid-CMOS step is a gate delay whereas an IMPLY, MAGIC, or MAD step is the application of a drive signal. For the approaches that use equivalent units, MAD offers the lowest step count across all Boolean operations.

A complete breakdown of the componential comparisons for the various Boolean operations is given in Table 5.

Table 5: Componential comparisons for memristor-based gates

Op	IMPLY	Hybrid-CMOS	MAGIC	Zhang	Threshold	MAD
NAND	3 memristors 3 drivers	2 memristors 2 MOSFETs	3 memristors	2 memristors	2 memristors 1 GOTO pair 2 MOSFETs	3 memristors 2 drivers
AND	4 memristors 4 drivers	2 memristors	3 memristors	2 memristors	2 memristors 1 GOTO pair	3 memristors 2 drivers
NOR	6 memristors 6 drivers	2 memristors 2 MOSFETs	3 memristors	2 memristors	2 memristors 1 GOTO pair 2 MOSFETs	3 memristors 2 drivers
OR	6 memristors 6 drivers	2 memristors	3 memristors	2 memristors	2 memristors 1 GOTO pair	3 memristors 2 drivers
XOR	7 memristors 7 drivers	6 memristors 2 MOSFETs	N/A	N/A	5 memristors 3 GOTO pairs 2 MOSFETs	3 memristors 2 drivers
NOT	2 memristors 2 drivers	2 MOSFETs	2 memristors	2 memristors	2 MOSFETs	2 memristors 2 drivers

In comparison to IMPLY gates, the MAD gates offer improved complexity for every Boolean operation. In comparison to the hybrid-CMOS gates, the MAD gates require more components for the AND and OR operations, but fewer for the remaining gates. Also, there is no need for signal restoration in circuits built from the proposed gate structure as required in hybrid- CMOS designs. This is because signals do not propagate through the circuits, but rather serve as sense voltages. For the gates which MAGIC and

Zhang, et al. have designs for, the proposed MAD gates have a slightly higher complexity. However, these works do not report the additional circuitry required for resetting, writing, and reading the memristors such as switches or comparators. They also do not report on the logic required for concatenating the gates. The area requirements for the threshold gate approach is comparable with the exception of the XOR gate, for which the MAD design has significantly fewer components. MAD designs offer further area savings when the same inputs are used for multiple gates by reusing the input circuitry. For example, performing A AND B and A OR B in parallel would require the input memristors A and B and two output memristors. Thus, for N gates using the same inputs A and B, the design only requires N+2 memristors rather than 3N for the other implementations. MAD gates also improve in energy consumption over prior work as shown in Table 6.

Table 6: Energy comparisons (J) for memristor-based Boolean gates

Operation	IMPLY	Hybrid-CMOS	MAD
NAND	7.1e-13	N/A	3e-14
AND	1.044e-12	1.75e-13	3e-14
NOR	1.044e-12	N/A	3.3e-14
OR	7.1e-13	1.75e-13	3.3e-14
XOR	3.03e-12	N/A	3.3e-14
NOT	3.4e-13	1.75e-13	3.3e-14

This energy was calculated by integrating V*I characteristics over the execution of the operation since the TEaM model does not have inherent power or energy measurements. The MAD gates improve energy by an order of magnitude over the IMPLY approach, mostly because of the large number of steps required for IMPLY operations. MAD gates require about 20% that of hybrid gates. The energy value for hybrid gates was computed by integrating the reported power and duration values in prior work.

Additionally, these designs do not suffer from the concatenation, parallelizability, and fanout challenges that the hybrid-CMOS, logic-in-memory, and MAGIC gate approaches have. Lastly, the MAD gates are currently able to be fabricated and modeled, rendering them a more practical option than GOTO-based threshold gate implementations.

Full Adder

Using MAD gates, it is possible to construct a full adder as shown in Figure 20.

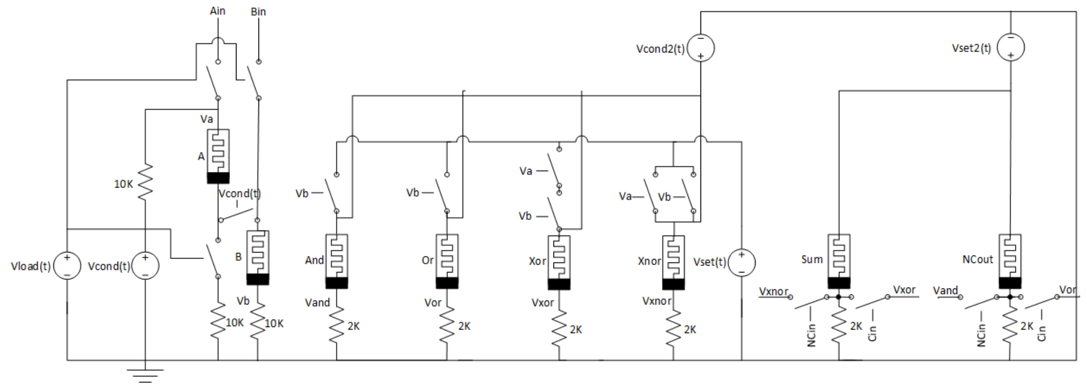


Figure 20: MAD full adder

At t=0, inputs A and B are initialized into the sense memristors by applying the V_{load} signal. When V_{load} is applied, memristors A and B are disconnected from each other and the inputs A_{in} and B_{in} are gated into the n terminals of the input memristors. For proper functionality, V_{load} must be greater than the threshold of the associated switches and the strength of A_{in} and B_{in} must be greater than V_{th} of the memristors. In this work, logical '1' for the inputs corresponds to V_{set}, V_{load}=V_{cond}, and V_{th}=1 V.

Next, the four memristors for the A OR B, A AND B, A XOR B, and A XNOR B operations are resolved in parallel, a key benefit of MAD methodologies. V_{cond} is applied to the input memristors in series and the voltages at V_a and V_b are used to drive the V_{set} signal to the output memristors as described for the gate operations. Lastly, the sum and inverse carry out results are resolved. In this step, V_{cond2} is applied to the intermediate results and V_{set2} is applied to the sum and inverse carry-out memristors. The carry-in signal and its inverse, C_{in} and NC_{in}, drive switches which gate the intermediate values to the p terminals of the sum and carry-out memristors. Essentially, the carry-in signal determines which intermediate result is used as the first parameter for IMPLY operations.

If the carry-in signal is a '1', then the operation (A or B) IMP 0 = NC_{out} executes else (A AND B) IMP 0 = NC_{out}. Since this is equivalent to a NOT operation, the corresponding equation is NC_{out} = Cin(A NOR B) or NC_{in}(A NAND B). Similarly, if the carry-in signal is a '1', then the operation (A XOR B) IMP 0 = Sum executes, else (A XNOR B) IMP 0 = Sum. Equivalently, Sum = C_{in}(A XNOR B) or NC_{in} (A XOR B). The carry-in strength and the associated switches follow the same voltage requirements as A_{in} and B_{in} and the input memristors. In all, this configuration has a delay of 2 and requires 8 memristors. If the drivers are included, the adder requires 8 memristors, 9 resistors, 5 drivers, and 14 switches.

Note that resolving the value of the inverse of the carry-out is as informative as resolving the carry-out. Since MAD designs are based on sensing nodes in the circuit, the value of a memristor and its inverse can be sensed in parallel by using the voltage at both the p and n terminal. This was demonstrated with the NOT and COPY operations in the earlier section. If the value of the carry-out signal is desired in a future step, then the n terminal of the NC_{out} memristor will be sensed when V_{cond} is applied, and if the inverse of the carry-out is desired the p terminal will be sensed. Thus, this adder not only resolves the sum and inverse carry-out signal, but also the inverse sum and carry-out signals.

The MAD full adder can be further optimized to reduce the delay of the full adder from 2 steps to just 1 after initialization and reduce the complexity to 4 memristors, 5 resistors, 13 switches, and 4 drivers. The circuit diagram for the proposed optimized MAD full adder is shown in Figure 21.

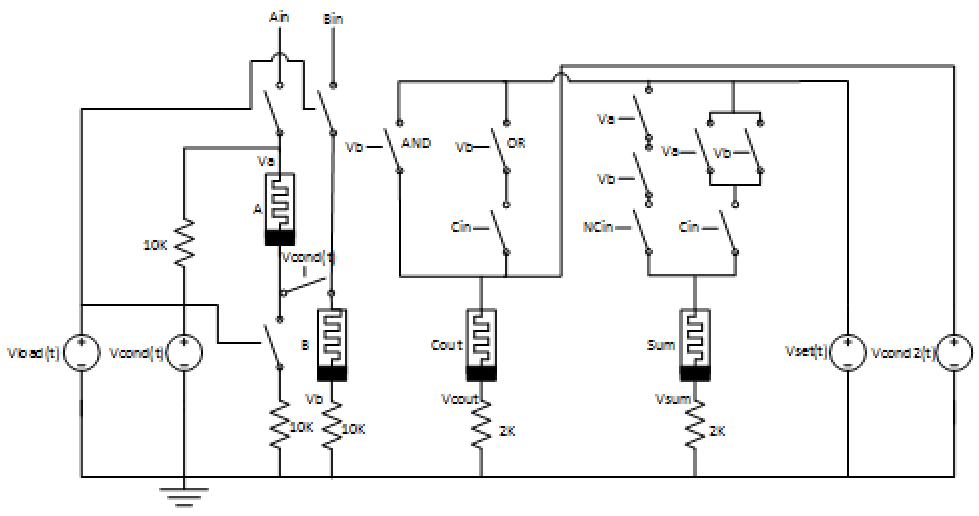


Figure 21: Optimized MAD-based full adder

Names and parameters were selected to match those in the previous figure to make the adaptations more clear. At t=0, the inputs are loaded into memristors A and B using the V_{load} signal as described in prior work. This initializes the adder with the input values. In the original work, intermediate results for A AND B, A OR B, A XOR B, and A XNOR B were computed in the next step and the carry out and sum values were resolved in the last. This design removes the need for the intermediate results, producing the carry-out and sum signals in a single step.

Similar to the original design, a read signal, V_{cond}, is applied to the input memristors in series and the voltages at nodes V_a and V_b are sensed to drive the switches on the neighboring memristors. However, at the same time these voltages are sensed, the value of the carry-in signal and its inverse are sensed. Depending on these values, the V_{set} signal is gated to the output memristors for carry-out and sum, setting them to a logical '1'.

Specifically, the circuitry for the carry-out memristor is implementing C_{out} = C_{in}(A or B) or AB. This is seen in the circuitry where the two paths from V_{set} to the carry-out memristor are. The left-most path will only gate V_{set} through to the memristor if the voltage at V_b is above the V_{th} for an AND operation. The right-most path will only gate V_{set} if the voltage at V_b is above the V_{th} for an OR operation and the carry-in signal is above the V_{th} of its switch. Now, if either AB or C_{in}(A or B), V_{set} will have a closed circuit through the carry-out memristor and the preceding Boolean equation is accomplished. A similar methodology is used for the sum memristor, implementing NC_{in}(A XOR B) or C_{in}(A XNOR B) via the left and right paths respectively. 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 Figure 19, and its inverse. Threshold voltages on the V_a and V_b switches are selected accordingly.

This approach to a MAD full adder reduces the need for the intermediate Boolean results by incorporating them into the circuitry for the output memristors themselves. This removes a step from the addition process and removes 1 driver, 4 memristors, 4 resistors, and 1 switch from the design.

IMPROVED AND OPTIMIZED ADDER DESIGNS[2]

This section presents completed work in the design and implementation of memristor-based adders. IMPLY, hybrid-CMOS, and threshold gate approaches to ripple carry, carry select, conditional sum, and carry lookahead adders are described and analyzed in terms of complexity and delay. Completed schematics and simulations are also given.

[2] L. Guckert and E. E. Swartzlander Jr., "Implementing Adders with Memristors," *ETCMOS Conference*, May 2016.
L. Guckert and E. E. Swartzlander Jr., "Memristor-based Threshold Gate Adders," *DAC Conference*, Jun. 2016.
This section is an extension of previously published work for which Lauren Guckert is the principal author and responsible for all content.

Ripple Carry Adders[3]

First, memristor-based ripple carry adders are presented for the IMPLY, hybrid-CMOS, threshold, and MAD contexts.

IMPLY Ripple Carry Adder

In this work, an optimized parallel implementation of an IMPLY ripple carry adder is presented to improve both delay and area. This work proposes a lower-latency, smaller area implementation than prior work, requiring 7N+1 memristors and a delay of 2N+19 steps. The complete execution steps are shown in Table 7.

Table 7: Steps for an optimized IMPLY parallel add operation

STEP	FUNCTIONALITY	GOAL
1	A imp 0	A->M2
2	B imp 0	B->M3
3	B	M3->M1
4	(A imp 0) imp (B imp 0)	M2->M3
5	A imp B	A->M1
6	((A imp 0) imp (B imp 0)) imp 0	M3->M0
7	B imp (A imp 0)	B->M2
8	B imp (A imp 0) imp 0	M2->M4
9	A XOR B	M1->M0
10		FALSE(M1,M2,M3
11	(A XOR B) imp 0	M0->M2
12	Cin imp ((A XOR B) imp 0)	Cin->M2
13	Cin AND (A XOR B) + AB-> **COUT**	M2->**M4**
14	*NEXT BIT IS READING CIN*	
15	Cin imp 0	Cin -> M1
16	C+ (A XOR B)	M1->M0
17	*NEXT BIT IS READING CIN*	
18	(C+ (A XOR B)) imp 0	M0->M3
19	(C NAND (AXORB)) imp ((C+ (A XOR B)) imp 0)	M2->M3
20		False M1
21	(C NAND (AXORB)) imp ((C+ (A XOR B)) imp 0) imp 0 = **SUM**	M3->**M1**

[3] 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]. This section is an extension of previously published work for which Lauren Guckert is the principal author and responsible for all content.

The implementation was designed to ensure that the C_{in} memristor was always used as the first operand in each IMPLY operation. This removes the need to copy the value of C_{in} before its use, removing 2 steps from the carryin-to-carryout critical path. Given this implementation, the delay for an N-bit adder is 11 cycles + 2*N + 8 = 2N+19 cycles.

The above requires input memristors A, B, and C_{in}, 3 working memristors, and 2 result memristors for C_{out} and the sum. The IMPLY driver circuitry is added to each memristor for applying V_{cond} and V_{set}. The full circuitry required for a 1-bit IMPLY adder is shown in Figure 22.

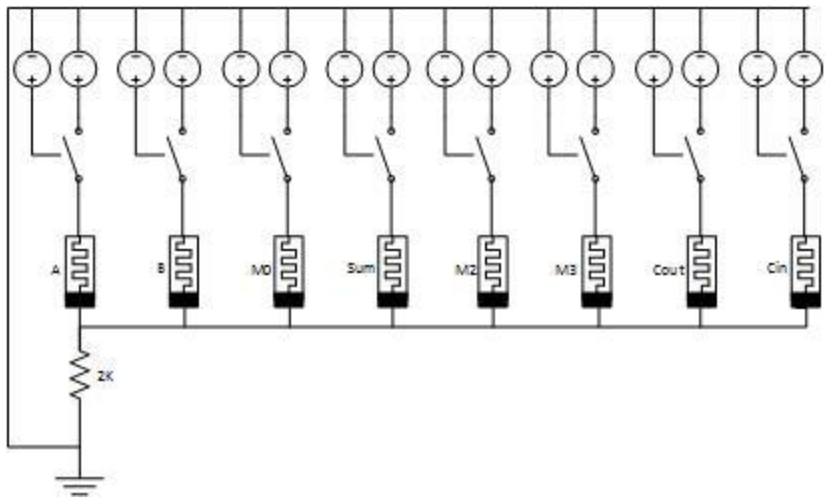


Figure 22: 1-bit IMPLY adder schematic

The result waveform for setting $A=B=C_{in}=1$ is shown in Figure 23. A cycle time of 1ns is used for ease of reading.

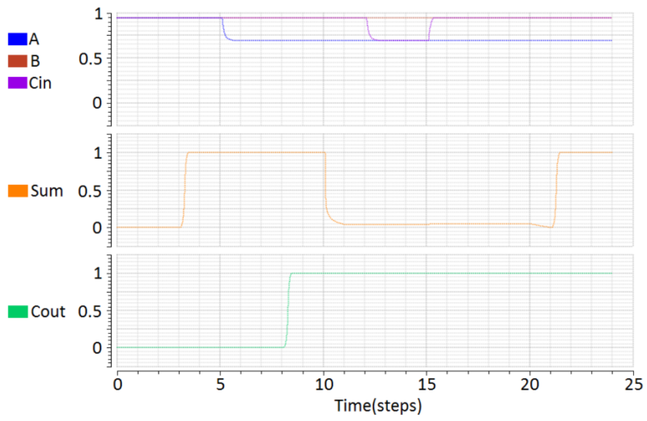


Figure 23: IMPLY adder execution for inputs A=B=C_{in}=1

The carry out bit is resolved in step 8 rather than the worst case of step 13 because both A and B are 1. This check is done in step 8, setting the carry out bit without needing the carry in bit. The sum bit is resolved in step 21 as confirmed by Table 7. Note that the signals degrade during execution. This is due to the operation 1 IMP 1, which degrades the strength of the first operand's "1" value. This can be corrected upon the next IMPLY operation.

For an N-bit adder, N of the full adders are placed in a chain. In order to allow individual bits to operate in parallel, there are voltage controlled switches on the lines between the bits. The full circuitry required for an 8-bit IMPLY adder is given in Figure 24.

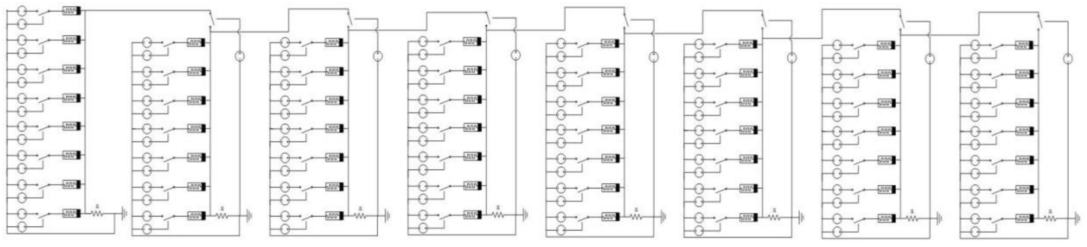


Figure 24: 8-bit IMPLY ripple carry adder

The design reduces the area as compared to previous designs by 2N memristors. This is due to strategic reuse of working memristors where is does not impact delay and the direct use of the memristor holding carry out bit N as the memristor for carry in bit N+1. The design reduces the delay from prior work 5N + 18 to 2N+ 19. 2N steps are saved by removing the need to copy the carry-in bits as described with Table 7. Additional reuse of intermediate values allows the critical path to reduce to 2, matching the critical path gate delay in traditional CMOS.

The functionality of an 8-bit adder implementation for worst-case inputs A=0xFF, B=0x00, and C_{in} = 1 is shown in Figure 25. Each ith carry bit is resolved 2 cycles after the (i-1)th sum bit, corresponding to the critical path of 2 indicated in Table 7.

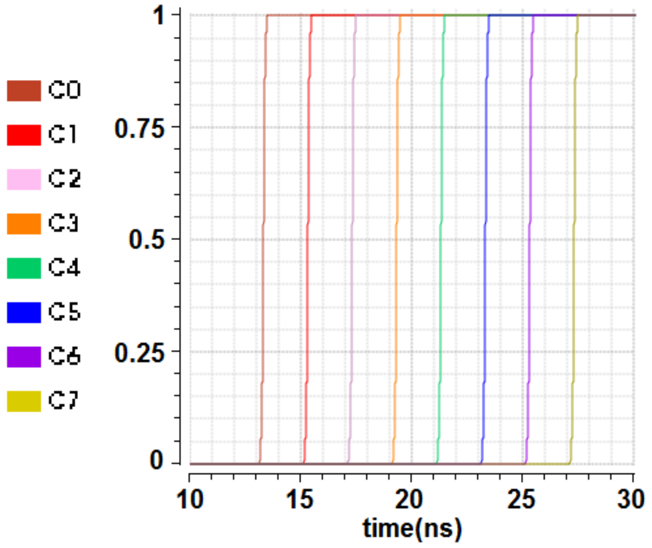


Figure 25: IMPLY adder execution for A=0x00, B=0xFF, C_{in}=1

An additional benefit of the IMPLY approach is that additions can be pipelined through the circuitry. Since the values are stored in the memristors and manipulated via drivers, each memristor can essentially be operated on disjointly, in parallel, as needed. In the current implementation, as soon as the sum bit i has been resolved, the circuitry for bit i sits idle until the next addition. Instead, consider beginning a new addition immediately. The overhead is two steps: 1 for resetting all of the memristors for bit i, and 1 for loading the next inputs A_i and B_i into their memristors for bit i. The only other modification to the execution steps in Table 7 is that the result of the sum should be stored in a separate memristor, outside of the adder. This step can now occur in step 20 since a result memristor does not need to be reset in bit i. In total, a new addition can begin every 22 steps in this IMPLY approach. This improves the throughput for the IMPLY ripple carry adder over alternatives and CMOS.

HYBRID-CMOS RIPPLE CARRY ADDER

This section presents the design of a novel hybrid-CMOS ripple carry adder optimized for area and delay via Boolean manipulation and an improved XOR gate implementation. The XOR gate is crucial to addition, occurring 2N times in the traditional ripple carry adder design. In this work, a new, lower-area, lower-delay hybrid-CMOS XOR circuit is proposed as shown in Figure 26.

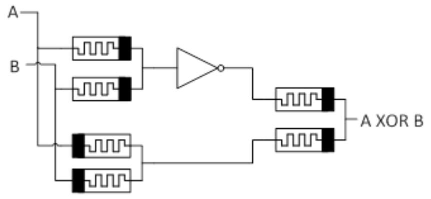


Figure 26: Hybrid-CMOS XOR gate schematic

This proposed circuit requires 1 fewer CMOS NOT gate (and 2 fewer vias) as compared to previous designs [20, 21] and maintains correctness. The result of the gate for each combination of inputs A and B is shown in Figure 27.

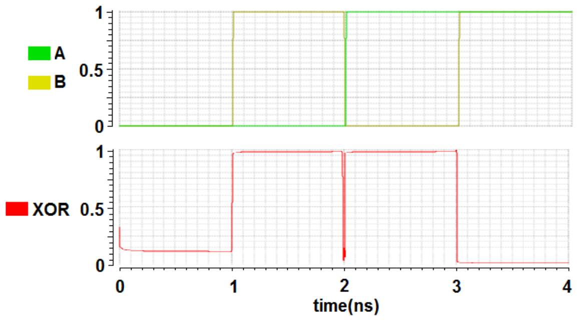


Figure 27: Hybrid-CMOS XOR gate execution

When used in a full adder, it further reduces the comparative latency and area because the intermediate A AND B result is calculated as part of the existing XOR circuitry whereas in other XOR designs it is not. This reuse can be seen twice in the schematic of a full adder in Figure 28.

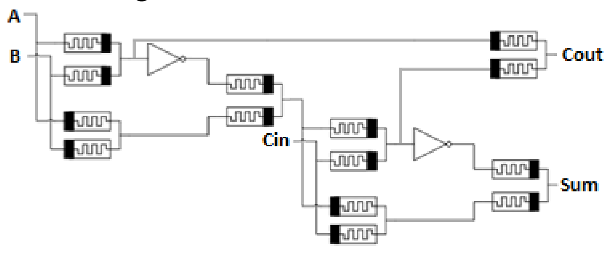


Figure 28: Hybrid-CMOS full adder based on the proposed XOR

This reuse removes 4 memristors from prior designs in addition to the savings from the proposed XOR gate. As the complexity of the arithmetic unit grows, these benefits become significant as shown in Table 8 below. Note the delay is the same for the proposed full adder.

Table 8: Area comparison for adders based on the proposed XOR gate

	Half Adder		**Full Adder**		**N-bit Adder**	
	Memristors	MOSFETs	Memristors	MOSFETs	Memristors	MOSFETs
Prior [20,21]	8	8	18	16	18N	24N*
Proposed	6	2	14	4	14N	12N*
CMOS		14		34		34N

*Includes 8N MOSFETs for buffering signals

At first glance, the delay of an N-bit hybrid-CMOS ripple carry adder is identical to CMOS, (2N+4) delays. The propagation of the carry signal takes 2 steps through each full adder, plus a constant delay for the first carry-out and final sum. However, one consequence of the hybrid-CMOS approach is that the signals suffer from signal degradation as they propagate through the adder. Thus, CMOS buffers must be added to the design to restore the signals with long propagation paths, such as the sum and carry out signals. The strength of the output signals before and after the buffers is shown in Figure 29 for each input combination for A, B, and Cin, starting with 000 and finishing with 111.

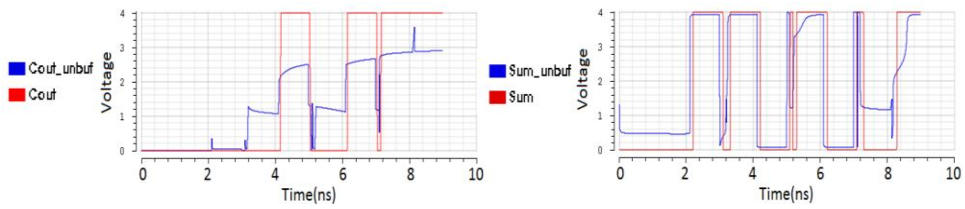


Figure 29: Hybrid-CMOS full adder execution without buffering (*_unbuf) and with buffering

The Cout_unbuf and Sum_unbuf signals represent the signals before the CMOS buffers, and the C_{out} and Sum signals represent the signals after the buffers. These buffers add 1 step to the delay and 8 MOSFETS to the area of each full adder. Thus, the total delay and complexity of an N-bit becomes 3N+4 and 14N memristors + 12N MOSFETs as shown in Table 8. This is still less than half the MOSFETs of the CMOS design and reduces the area from prior work by 4N memristors and 12N MOSFETs. The final gate-level schematic is shown in Figure 30.

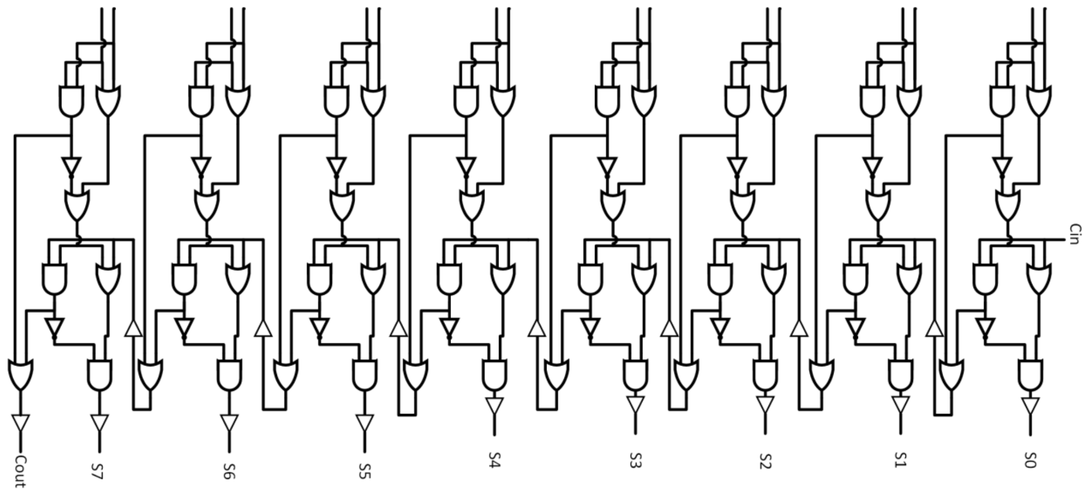


Figure 30: 8-bit Hybrid-CMOS ripple carry adder

The functional waveform for an 8-bit hybrid-CMOS adder with buffering is shown in Figure 31 for the worst case input combination: A=0x00, B=0xFF, C_{in}=1.

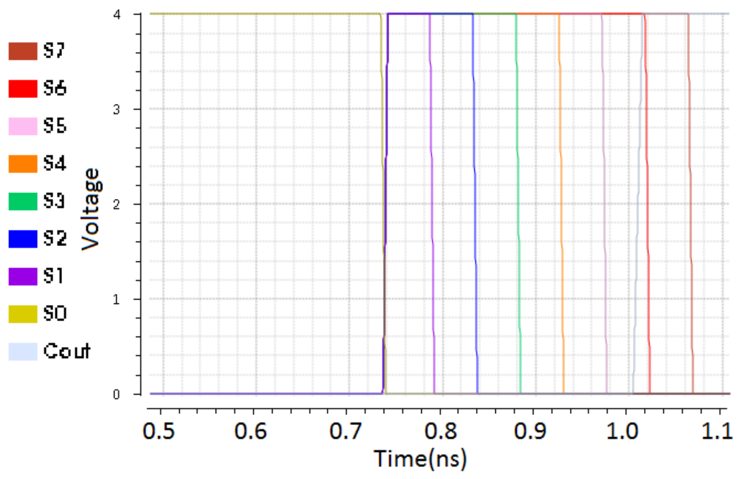


Figure 31: Hybrid-CMOS adder execution for A=0x00, B=0xFF, C_{in}=1

The sum bits resolve 3 steps apart and the carry out bit is resolved a cycle sooner than the last bit. Each sum bit rises to 1 after resolving the A and B inputs. Upon arrival of the respective C_{in} bit, the sum bit drops back to zero.

THRESHOLD GATE RIPPLE CARRY ADDER

A third implementation of an N-bit ripple carry adder uses threshold gates as the fundamental block. These threshold gates can be constructed from either GOTO pairs or CSTG gates.

These approaches leverages the Kautz/Minnick methods for constructing 3-to-2 counters from threshold gates [27]. With this, a full adder requires only 2 threshold gates as shown in Figure 32 for the Goto implementation. The delay for C_i to Ci_{+1} is 1 gate delay and the delay for C_i to S_i is 2 gate delays so the delay for an N-bit ripple carry adder is N+1 threshold gate delays. If GOTO pairs are used, the delay is then N+1 GOTO delays with a complexity of 7N memristors and 2N GOTO pairs. If CMOS buffers are used, the delay is (3N + 3) gate delays and the complexity is 7N memristors, 7N resistors, and 22N MOSFETs.

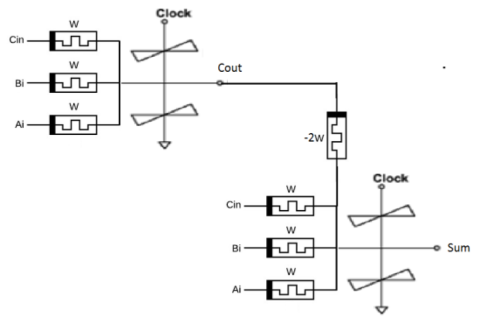


Figure 32: Threshold gate full adder using GOTO pairs

Figure 33 shows an 8-bit adder constructed from the proposed threshold gate full adders. For a threshold gate TGx, the output is a logical '1' when the inputs sum to greater or equal to x and a logical '0' when they sum to less than x.

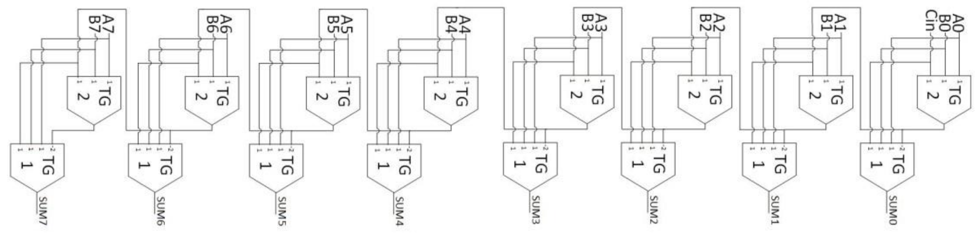


Figure 33: Optimized threshold gate 8-bit adder

MAD GATE RIPPLE CARRY ADDER

A MAD full adder can be extended with minimal circuitry to implement an N-bit ripple carry adder as shown in Figure 34.

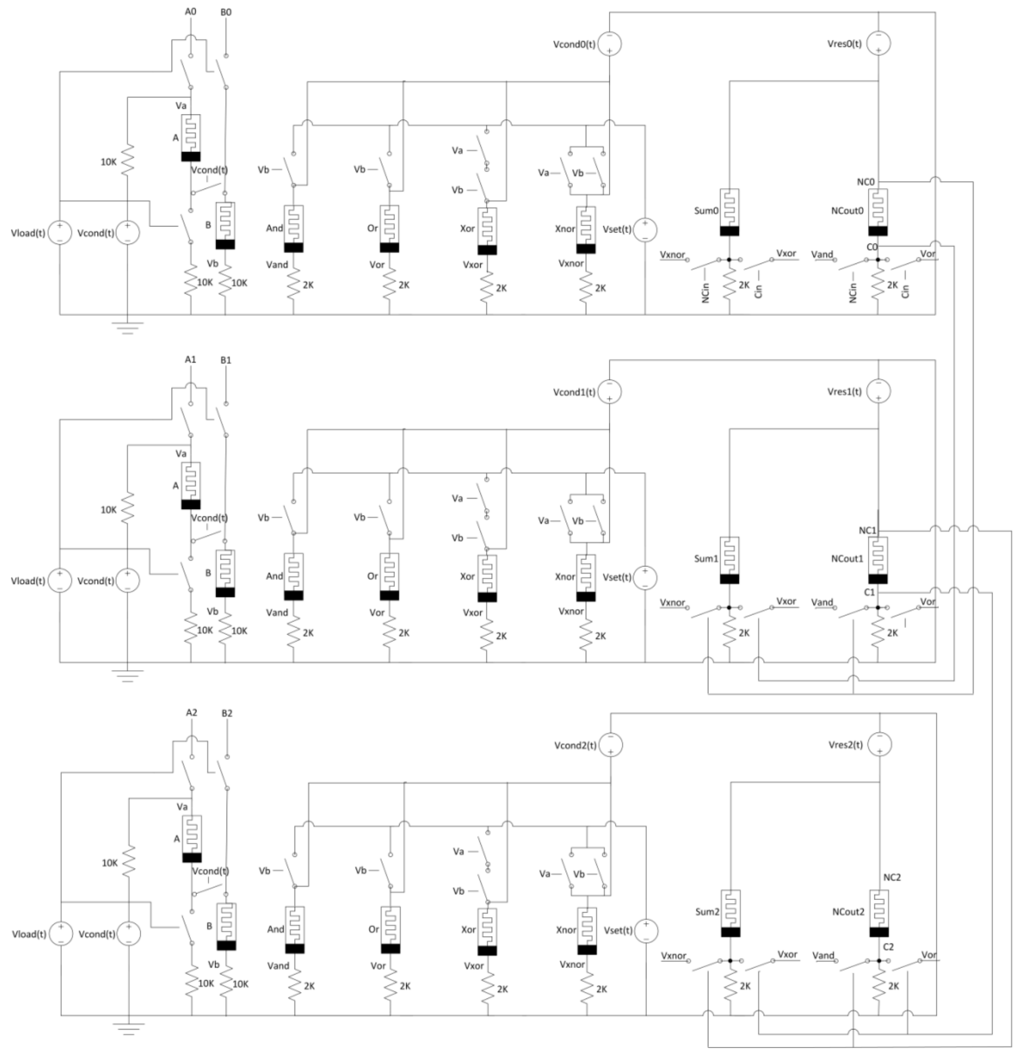


Figure 34: 3-bit MAD ripple carry adder

Figure 34 shows a 3-bit MAD ripple carry adder, but any adder width can be constructed by concatenating full adders in the same manner. The total complexity is 8N memristors, 9N resistors, 14N switches, and 2N+3 drivers. The load and intermediate operation steps can now be performed in parallel across the individual bits. At t=0, the V_{load} driver for the input memristors is applied to each bit as described for the full adder, loading the inputs A_i and B_i into their corresponding full adders. Then, all of the intermediate Boolean operations can be resolved by applying V_{cond} and V_{set} respectively. Lastly, each of the bits can resolve their sum and carry signals one step at a time. First, bit 0 resolves its sum and inverse carry-out. In the next cycle, the inverse carry-out memristor is applied a V_{cond} signal so that the voltage at the n terminal represents the value of the carry-out signal and the voltage at the p terminal represents the inverse. These two voltage strengths drive the gates on the subsequent bit in the same way the carry-in bit did for the full adder. The process then repeats for the subsequent bits until the final sum and carry-out are resolved. An N-bit addition requires N+1 steps as shown in Figure 35.

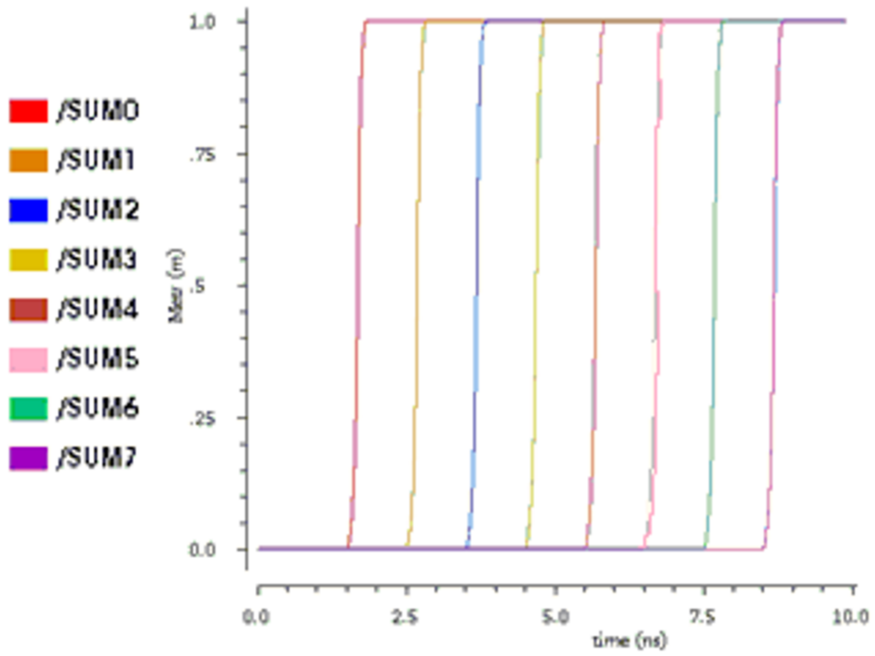


Figure 35: 8-bit MAD ripple carry addition for 0x0 + 0xFF

This N-bit ripple carry adder was then implemented with the optimized MAD full adder as shown in Figure 36.

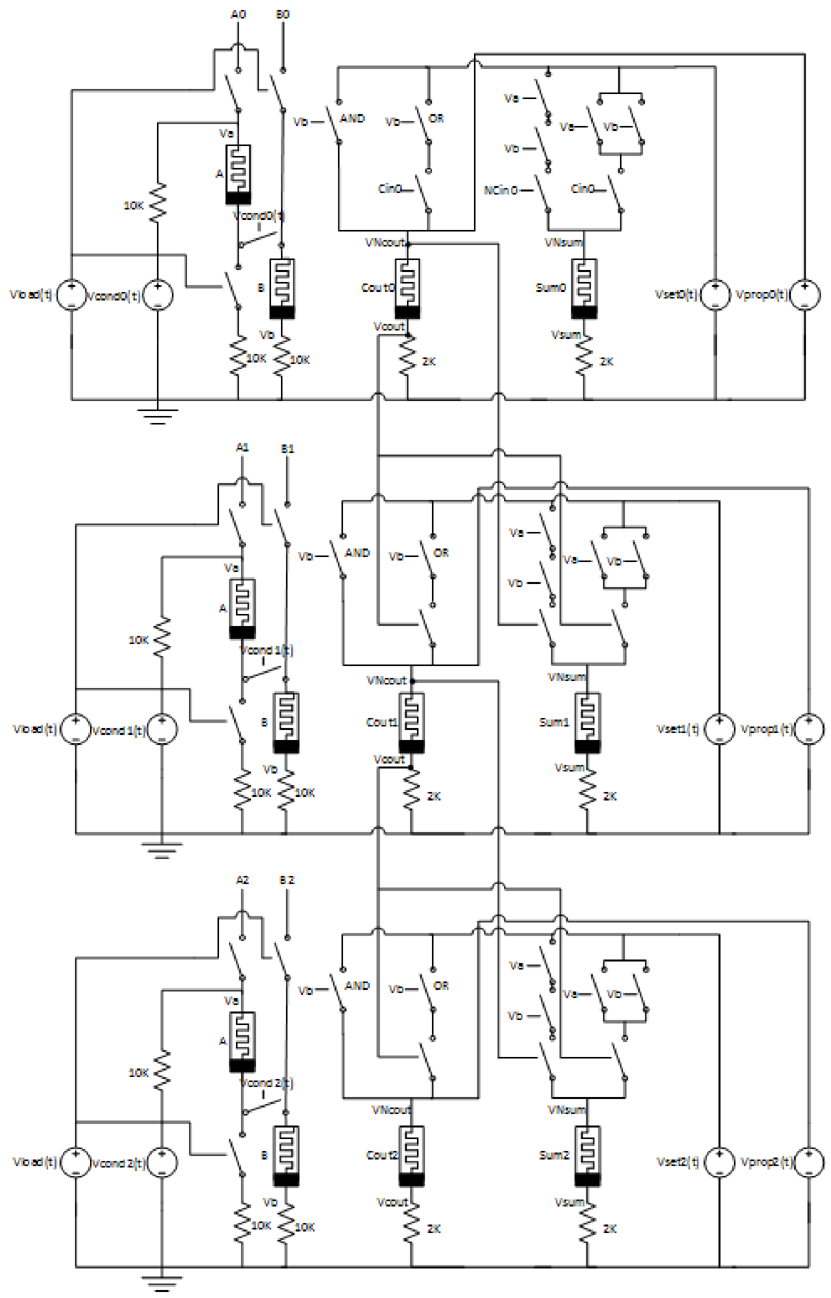


Figure 36: Optimized MAD design for an N-bit ripple carry adder

The V_{load} driver loads the inputs Ai and Bi into every full adder's input memristors at t=0. Then, in the first step, bit 0 resolves its Carry-out and Sum memristor as done in the original MAD design. In the next step, V_{prop2} applies a read signal, V_{cond}, to the C_{out} memristor. Then, as done in the original design, the n terminal can be sensed to represent the inverse of the carry-out and the voltage at the p terminal can be sensed to represent the carry-out signal. These values are then used in the same manner as the first full adder to drive the respective carry in signals on the next adder. The process repeats through each bit of the adder, resolving the final carry-out and sum in the Nth step. This is one less step than the original MAD design.

The total complexity for the N-bit adder is 4N memristors, 5N resistors, 13N switches, and 3N+1 drivers. The number of drivers is fewer than the number of memristors because the V_{load} driver has identical behavior for all bits and can be reused across them. The area and delay comparisons with the original MAD adder are summarized in Table 9.

Table 9: Delay and complexity improvements for the proposed MAD adders

	Original MAD Full Adder	Proposed MAD Full Adder	Original MAD RCA	Proposed MAD RCA
Complexity	8 memristors 9 resistors 14 switches 5 drivers	4 memristors 5 resistors 13 switches 4 drivers	8N memristors 9N resistors 14N switches 2N+3 drivers	4N memristors 5N resistors 13N switches 3N+1 drivers
Delay	2	1	N+1	N

The proposed MAD design can further improve delay by pipelining independent additions, similar to the IMPLY approach. Since the computations on the bits are performed through the use of drivers, the bits can be logically separated. Thus, as soon as bit 0 has completed its portion of an addition and its carry out has been sensed by bit 1, it can begin a new addition in parallel with the remaining bits finishing the current addition. The overhead is 2 steps: 1 for sensing the result, and 1 for resetting the memristors. However, since each memristor in the adder is disjoint, the memristors can each be reset during other steps of the addition, reducing the overhead to a single step. Thus, every 4 cycles a new addition can begin. In traditional CMOS and the other approaches, additions cannot overlap and must wait for the previous addition to complete.

RIPPLE CARRY ADDER ANALYSIS AND COMPARISON

A summary of the area and delay requirements for the presented approaches and traditional CMOS is given in Table 10.

Table 10: Complexity and delay comparisons for the ripple carry adders

	CMOS	IMPLY	Hybrid-CMOS	Threshold	MAD
Delay	2N+4	2N+19	3N+4	N+1	N
Complexity	34N MOSFETs	7N+1 memristors 7N drivers 8N-1 switches N resistors	14N memristors 12N MOSFETs	7N memristors 2N GOTO pairs	4N memristors 3N+1 drivers 13N switches 5N resistors

In terms of delay, all proposed implementations have comparable or better results relative to the CMOS approach. The IMPLY methodology maintains a delay of two for carry propagation (same as CMOS), and GOTO pairs and MAD gates improve this to 1. When buffering is considered for the hybrid CMOS design, the delay is increased to 3 steps per carry propagation for a total delay of 3N+4. However, the switch time is faster for memristors, making the 3 gate delay of the hybrid approach likely comparable to the 2 CMOS gate delays.

In terms of area, all discussed implementations improve upon CMOS\ The proposed hybrid-CMOS design has about a 38% reduction in the number of devices as compared to previous work. The proposed IMPLY implementation decreases the area from the prior state-of-the-art count of 9N memristors to 7N+1, significantly decreasing the number from 34N MOSFETs in CMOS. The GOTO-based designs see one of the greatest improvements in area, decreasing the requirements from 34N MOSFETs to 7N memristors and 2N GOTO pairs. The MAD design requires only 4N memristors, 5N resistors, 3N+1 drivers, and 13N switches as compared to 34N MOSFETs for a traditional CMOS design. This is fewer components than the hybrid-CMOS approach and does not require CMOS inverters. The MAD approach requires slightly more components than the IMPLY approach: the approach generally chosen for its low area.

Recall that MAD and IMPLY implementations have the additional benefit of overlapped operations. Since their functionality is controlled by drivers, they are able to implement various stages of numerous operations in parallel, pipelining executing and improving throughput as shown in Table 11.

Table 11: Throughput comparisons for X additions in ripple carry adders

	CMOS	IMPLY	Hybrid-CMOS	Threshold	MAD
Delay	2NX+4X	2N+22X-3	3NX+4X	NX+X	N+4X-4

In the MAD design, a new addition can begin every 4 steps. In IMPLY, a new addition can begin every 22 steps. In traditional CMOS and the remaining approaches, a new addition cannot begin until the full execution has completed on the previous addition. Thus, the throughput of the IMPLY and MAD adder design is significantly higher than alternative designs, performing X additions in X+O(N) rather than O(NX). As the number of additions increases, the MAD design establishes a significant advantage over the IMPLY approach. As X increases, the MAD design has a total latency of 4X, less than 1/5 of the IMPLY latency, 22X. One consequence of this optimization for the MAD designs is that the drivers cannot be shared across bits, increasing the number of drivers from 3N+1 to 5N.

Carry Select Adders

Memristor-based carry select adders are presented for the IMPLY, hybrid-CMOS, threshold, and MAD contexts.

IMPLY Carry Select Adder

Carry select adders can be highly optimized for the IMPLY context through Boolean translation. For example, when AND and OR IMPLY operations are known to perform consecutively, the total operation requires only 1 extra memristor and can reduce the necessary IMPLY steps to 3 rather than 7. In carry select adders, the carry signal propagation is expressed by CP+G, where P is the propagate signal, G is the generate signal, and C is the carry in signal. This can be simplified to:

1. P imp 0
2. C imp (P imp 0)
3. (C imp (P imp 0) imp G = CP+G

The first step can be hidden when the propagate signal is available before the carry in bit which is true for all blocks in the carry select adder besides the first two blocks. This makes the step delay of the IMPLY logic equivalent to CMOS Boolean logic.

Carry select adders also use multiplexers extensively. The equation for a multiplexer is optimized for the context of traditional CMOS Boolean gates and their transistor complexity. However, the same tradeoffs do not exist in the memristor domain for IMPLY operations. Thus, the multiplexer equation is rewritten to an equivalent form.

$$(A \text{ AND } S) \text{ OR } (B \text{ AND } \overline{S}) == \overline{(B \text{ NAND } \overline{S})} \text{ OR } \overline{(A \text{ NAND } S)} = \text{Out}$$

By interpreting the equation for the multiplexer as such, the number of steps is reduced. The execution steps for the optimized multiplexer are given in Table 12.

Table 12: Steps for an optimized 1-bit IMPLY multiplexer

$(\overline{B \text{ NAND } \overline{S}}) \text{ OR } (\overline{A \text{ NAND } S}) = \text{Out}$

Step	Mux Functionality	Mux Goal
1	A IMP $\overline{S}$	A NAND S
2	B IMP S	B NAND $\overline{S}$
3	(A IMP $\overline{S}$) IMP Out	A AND S
4	(B IMP S) IMP Out	(A AND S) OR (B AND $\overline{S}$) = Out

S represents the select line memristor and $\overline{S}$ represents the memristor holding the inverse of the select line. Note that this series of steps assumes that the memristors have been initialized ahead of time with a single write operation (as described in the previous section). First, A NAND S is computed. Next, B NAND $\overline{S}$ is computed in the same manner. Finally, the two outputs are OR'ed together to produce the output. The equivalent schematic is shown in Figure 37. It requires 2 input memristors, A and B, 2 memristors, S and its inverse for the select lines, and 1 output memristor, Out for a total of 5 memristors. No internal working memristors are necessary.

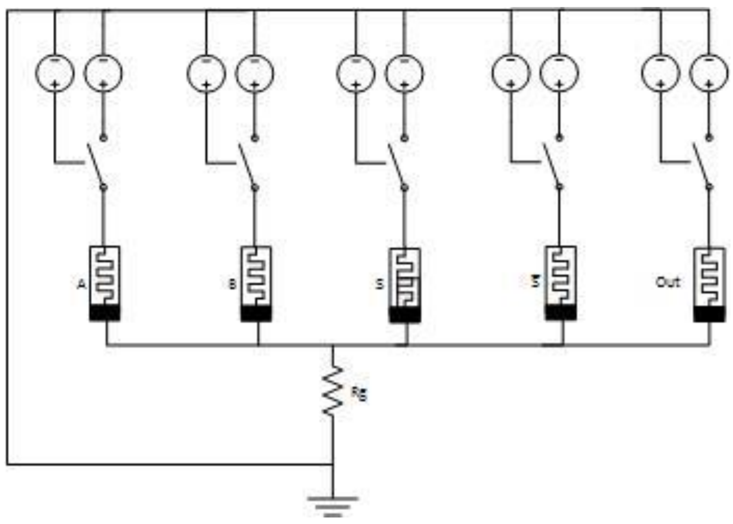


Figure 37: IMPLY Multiplexer

When used in the context of the carry select adder, the A and B memristors are not necessary as they exist as output memristors in the ripple carry adders. This reduces the complexity of each multiplexer to 3 memristors. The waveform for the execution of the multiplexer when A=1, B=0, and Sel=1 is shown in Figure 38.

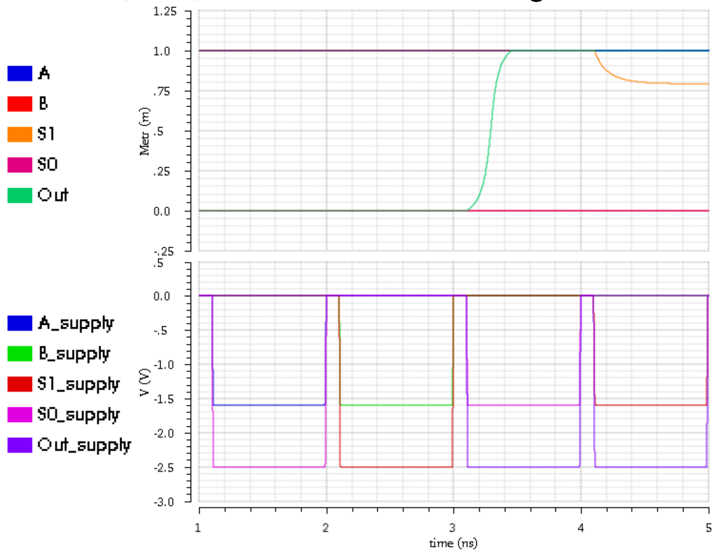


Figure 38: IMPLY multiplexer execution

The lower waveform shows the corresponding V_{cond} and V_{set} signals in each cycle to align with Table 12. The resultant memristor values are shown in the upper graph, where the Out value is resolved in step 3.

Traditionally, in CMOS implementations, the optimal block width k (in terms of minimizing delay) is equivalent to the $\sqrt{N}$ for an N-bit adder. However, the IMPLY implementation has different delay characteristics and thus could have a different optimal block width. If a constant block width of k is used, where N is an integer multiple of k, N/k blocks will exist. An IMPLY based ripple carry adder has a delay of 2k+19 for the sum, the carry out is 2k+11 steps, and the complexity is 7k+1 memristors. So for an N-bit carry select adder, it takes 2k+11 steps to form the carry out of the first block, 3 steps for the second block, 2 steps for each of the N/k –3 intermediate blocks, and 6 steps in the final block, dependent on whether the sum bits arrive before the carry in bit.

The intermediate blocks take one less delay than the second block's propagation because the (P imp 0) operation can be performed after the P bit arrives at 2k+11 steps, prior to the carry in bit arriving. Note that since the carry-in memristors are not overwritten by the carry propagation, they do not need to be set and sensed during these stages. They only need to be set and sensed upon use in the multiplexers which can occur in a later, idle step. The carry-in bit propagation operations do not overlap with the sum calculations of prior bits. Thus,

$$STEPS_{CSEL} = 2k + 14 + 2(N/k) \qquad (8)$$

where $STEPS_{CSEL}$ is the total delay. The optimum block size is found by taking the derivative of $STEPS_{CSEL}$ with respect to k and solving for k. The result is

$$k = \sqrt{N} \qquad (9)$$

$$STEPS_{CSEL} = 4\sqrt{N} + 14 \qquad (10)$$

This is slightly higher than the traditional CMOS design which has a delay of 4√N+2. However, as the size of the adder increases, the impact of the constant in the delay equation decreases and they become roughly equivalent. The proposed design's delay is significantly less than the optimized ripple carry adder (2N+19). For a 16-bit adder, the comparison in delay is 30 for the carry select adder vs. 51 for the ripple carry adder and for a 64-bit adder the difference in delay increases to 46 vs. 147. A waveform for the worst case 16-bit addition, A=0xFFFF, B=0x0000, and C_{in} = 1 is shown in Figure 39.

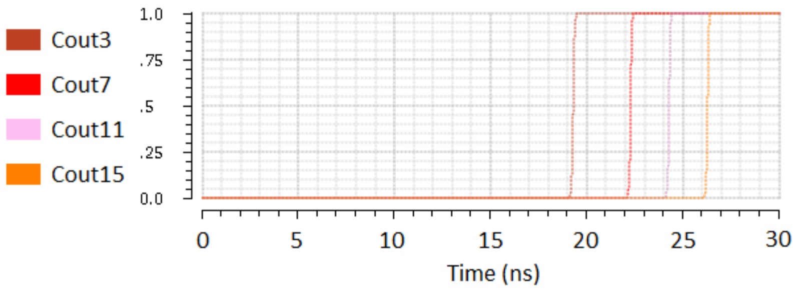


Figure 39: 16-bit IMPLY carry select adder execution for A=0xFFFF,B=0x0000, and C_{in}=1

Each ripple carry generates its output at 2(4)+11 steps, or t=19n as shown by C_{out}3. The next block has a 3 step delay since the first step of carry propagation (P imp 0) cannot be performed a priori. The remaining blocks have a carry propagation delay of 2 steps.

The carry select adder can be pipelined in a similar manner to the IMPLY ripple carry adder. Once the sum and carry-signals have been resolved and propagated for a given bit, the circuitry can be reset and reused for a subsequent addition. Thus, every 22 steps a new addition can be started.

The schematic for this carry select adder is in Figure 40.

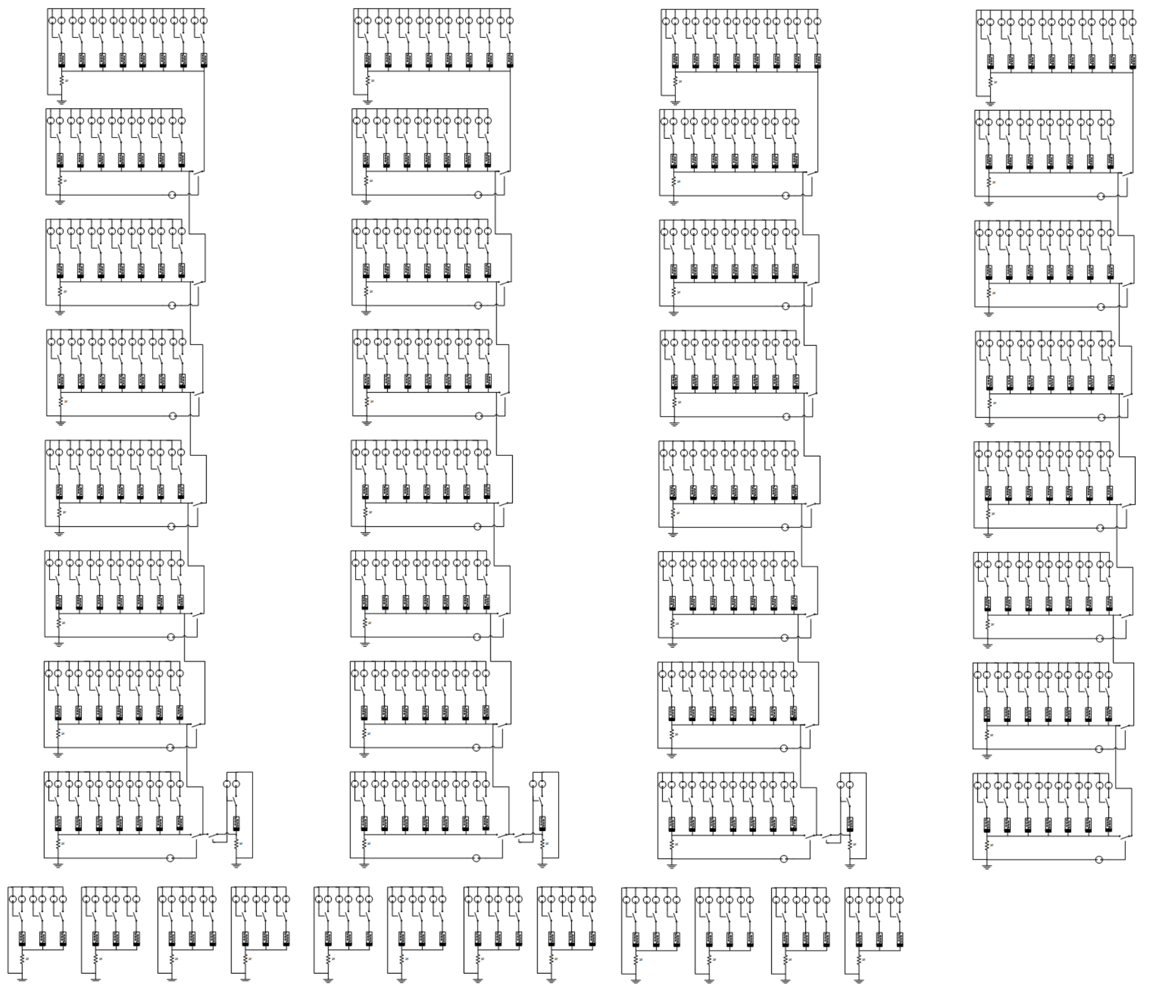


Figure 40: 16-bit IMPLY carry select adder schematic

The complexity of the carry select adder is 7(2N -k)+2N/k-1 memristors for the ripple carry adders, N/k - 2 memristors for the AND and OR gate propagations, and 3(N – k) for the sum multiplexers.

$$\mathrm{MEMRISTOR}_{\mathrm{SCSEL}} = 17N - 10k + 3N/k - 3 \qquad (11)$$

This result is around 2.5x the complexity of a ripple carry adder depending on the selection of k. For $k=\sqrt{N}$, the number of memristors becomes $17N-7\sqrt{N}-3$ or 241 for a 16-bit adder. Note that the carry propagation logic accounts for only N/k-2 memristors in the total design, 2 memristors in the 16-bit case.

Hybrid-CMOS Carry Select Adder

A carry select adder implemented with the Hybrid-CMOS approach is also optimized and analyzed. A schematic for a multiplexer in a hybrid-CMOS design is shown in Figure 41.

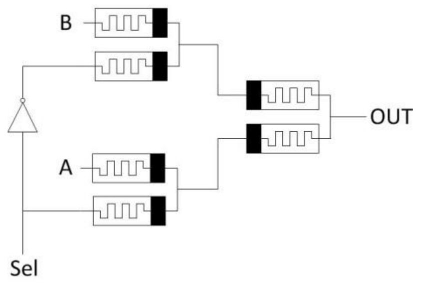


Figure 41: Hybrid-CMOS multiplexer

Its delay in terms of the number of gates is the same as traditional CMOS. The complexity is also the same at a high level, 1 NOT gate, 2 AND gates, and an OR gate. This translates to fewer components than in traditional CMOS: 2 MOSFETs, 6 memristors, and 2 vias as opposed to 17 MOSFETs. For an N-input mux, the complexity is 2 MOSFETs + 6N memristors + 2 vias as compared to 15N + 2 MOSFETs in traditional CMOS. However, the outputs must be buffered in order to restore signal strength. This adds 4N MOSFETs to an N-bit mux. Fortunately, this buffering also produces the inverse output which can be leveraged to remove delays further in the overall design as discussed later.

The delay for an N-bit adder with block size k consists of the delay through the ripple carry adders, carry propagation, and the final sum multiplexer. In the hybrid approach, it takes 2k+3 steps to form the carry out of the first block, 4 steps for each of the N/k –2 intermediate blocks, and 2 steps in the final block. Thus,

$$\text{STEPS}_{\text{CSEL}} = 2k - 3 + 4(N/k) \tag{12}$$

Which simplifies to $\text{STEPS}_{\text{CSEL}} = 6\sqrt{N}-3$ for the optimized block size $k=\sqrt{N}$. For a 16-bit adder, this delay is 21 vs. 30 for the IMPLY implementation discussed in the previous section, reaping about a 1/3 improvement. A waveform for the worst case addition where A=0xFFFF, B=0x0000, and C_{in}=1 is shown in Figure 42.

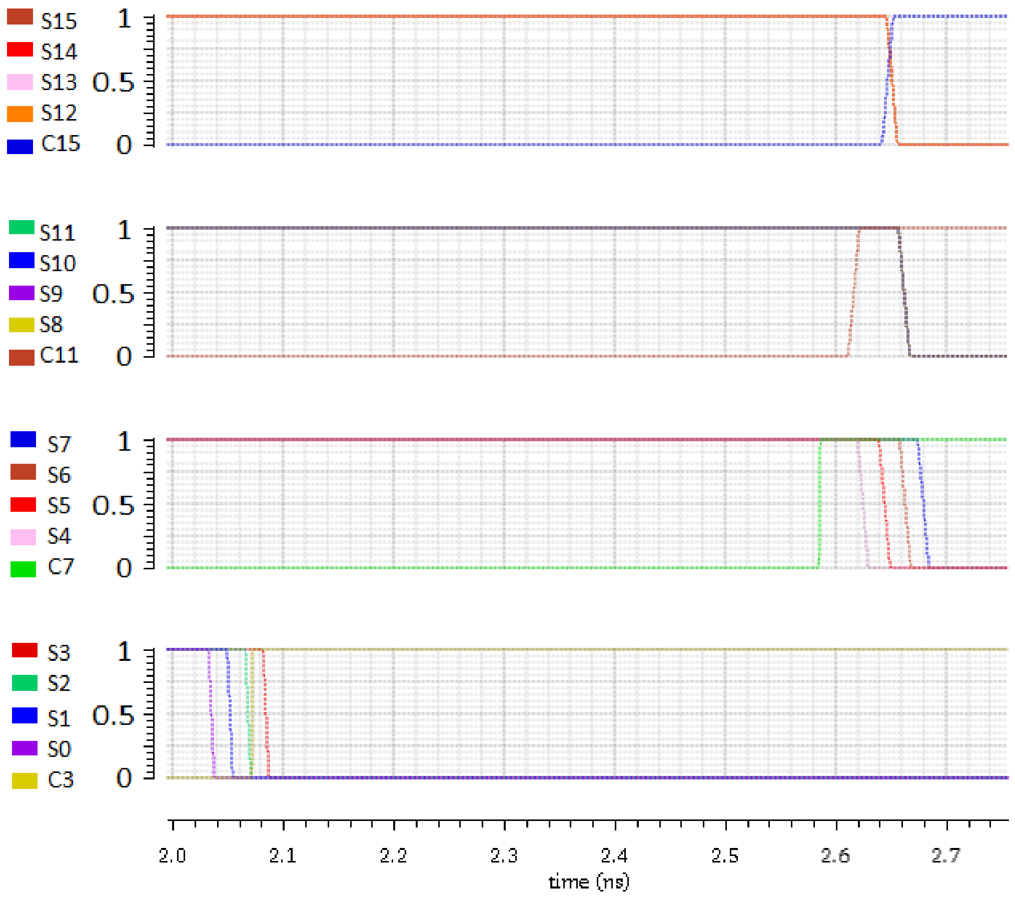


Figure 42: Hybrid-CMOS carry select adder execution for inputs A=0xFFFF,B=0x0000, C_{in}=1

Some optimizations have been performed to minimize this complexity of the overall design. For example, the design takes advantage of the fact that each ripple carry adder and carry propagation circuit has buffered outputs. This means that the inverse of every carry signal is produced as part of the buffer process and does not need to be included in the multiplexer designs. Thus, the total complexity is (2N -k)(14 memristors + 16 MOSFETs + 9 vias) for the ripple carry adders, (N/k-1)(6k memristors) for the multiplexers, and (N/k-2)(4 memristors+4 MOSFETs+3 vias) for the carry propagation.

$$\begin{aligned} \text{COMPLEXITY}_{\text{SCSEL}} = {} & (34N-20k+4N/k-8) \text{ memristors} \\ & + (32N-16k+4N/k-8) \text{ MOSFETs} \\ & + (18N-9k+3N/k-6) \text{ vias} \end{aligned} \tag{13}$$

For a 16-bit adder, this becomes 472 memristors + 456 MOSFETs + 258 vias. This is over twice the complexity of the proposed IMPLY method and involves multiple technologies. The full schematic for this implementation is shown in Figure 43.

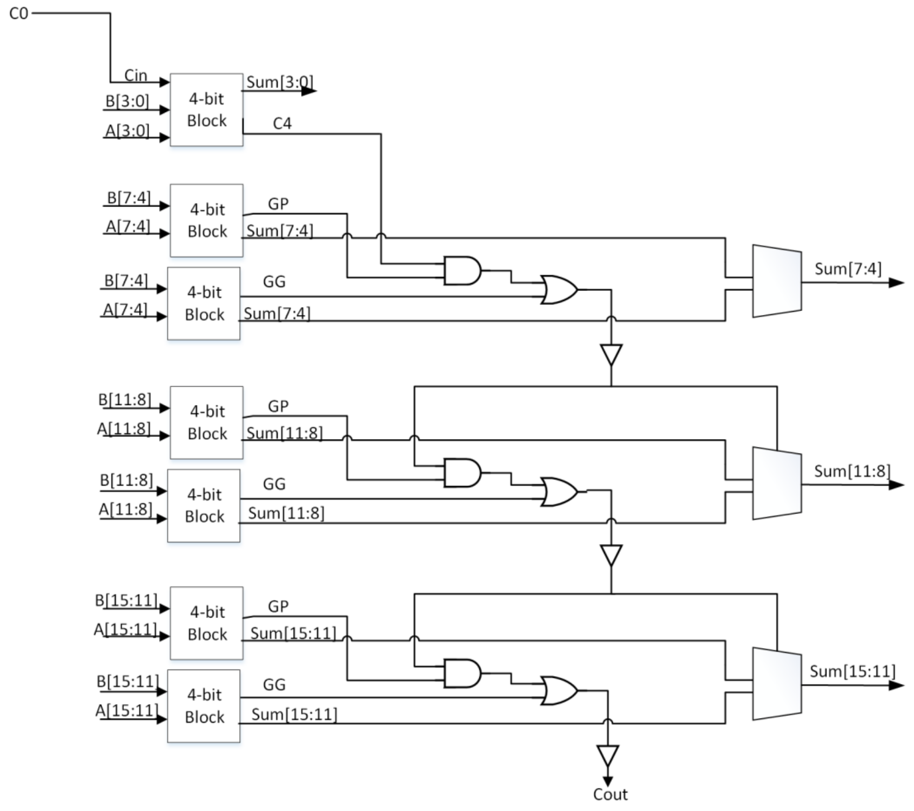


Figure 43: Optimized hybrid-CMOS carry select adder schematic

THRESHOLD GATE CARRY SELECT ADDER

Carry select adders using threshold gates are proposed, optimized, and analyzed for both GOTO based and CSTG-based designs.

The carry propagation logic equation, G+PC, can be implemented with a single threshold gate by applying the weights {P:1, C:1, G:2} to the gate inputs. A multiplexer can be implemented using 2 threshold gates and a NOT gate as shown in Figure 44.

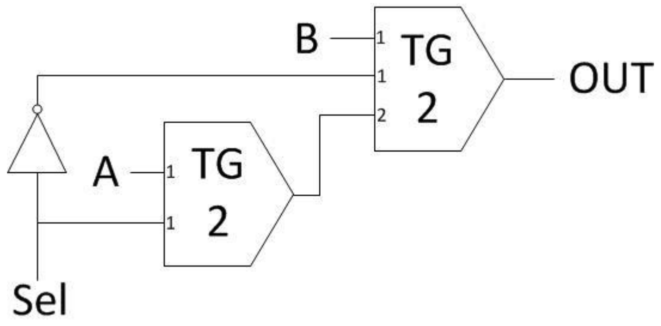


Figure 44: Threshold gate multiplexer

This is the cheapest design in terms of delay because the select line (in this case the carry-in bit) must be inverted, incurring a single delay. A b-bit mux requires 2b+1 gates. If the GOTO threshold implementation is selected, it is assumed that an all-memristor-based design is preferential, leading to an IMPLY based implementation of the NOT gate. For a b-bit mux, this has a complexity of 5b+1 memristors and 2b GOTO pairs with a delay of 2. If the CSTG threshold implementation is selected, it is assumed that a hybrid-CMOS memristor approach is acceptable and the NOT gate is implemented using standard MOSFETs. For a b-bit mux, this results in a complexity of 18b+2 MOSFETs, 2 vias, 5b memristors, and 5b resistors, and a delay of 2.

For GOTO implementations, the ripple carry adders account for (2N/k -1)(7k memristors, 3k+2 vias, and 2k GOTO pairs) or 14N-7k memristors + (6N-3k+4N/k-2) vias + 4N-2k GOTO pairs. The multiplexer logic costs (N/k-1)(5k+1 memristors + 2k GOTO pairs) or 5N-5k+N/k-1 memristors + 2N-2k GOTO pairs. The carry propagation logic costs (N/k-2)(3 memristors and 1 GOTO pair) or 3N/k-6 memristors and N/k-2 GOTO pairs. The total complexity is (19N-12k+4N/k-7) memristors + (6N-3k+6N/k-4) vias + (6N-4k+N/k-2) GOTO pairs. The total delay is (k+1) + (N/k-2) +2 = k+N/k+1 GOTO delays.

For CSTG implementations, the ripple carry adders account for(2N/k-1)(7k memristors + 22k MOSFETs + 7k resistors + 3k+2 vias) or 14N-7k memristors + 4N-22k MOSFETs + 14N-7k resistors + 6N-3k+4N/k-2 vias. The multiplexers cost (N/k-1)(18k+2 MOSFETs + 2 vias + 5k memristors + 5k resistors) or (18N-18k+2N/k-2) MOSFETs + 2N/k-2 vias + 5N-5k memristors + 5N-5k resistors. The carry propagation logic costs (N/k-2)(3 memristors + 3 resistors + 10 MOSFETs) or 3N/k-6 memristors + 3N/k-6 resistors + 10N/k-20 MOSFETs. The total complexity is 19N-12k+3N/k-6 memristors + 22N-40k+12N/k-22 MOSFETs + 19N-12k +3N/k+6 resistors + 6N-3k+6N/k-4 vias. The total delay is k+N/k+1.

MAD GATE CARRY SELECT ADDER

A MAD carry select adder is presented to optimize delay and area for the MAD context. The k-bit adders take roughly the same form as the MAD ripple carry adder. The first k-bit block is identical to a standard MAD ripple carry adder since the carry-in signal is known at time=0. The remaining k-bit blocks have a slightly modified form. The properties of the MAD full adder can be leveraged to remove the need for the second adder in each adder block. The MAD full adder achieves its low delay by producing the intermediate results for the scenarios where the carry-in is 0 and 1. Thus, the information which spans two ripple carry adders in the traditional design is encapsulated naturally in a single adder in MAD gates. The only adaptation is that the final result memristor circuitry for the sum and carry-out memristors must be duplicated. The first sum and carry-out memristor will be applied the Carry-in according to the scenario where the carry-in to the block is 0 and the second sum and carry-out memristor will be applied the carry-in according to the carry-in to the block being 1. These additions can be performed in parallel independent from each other. The full adder is shown in Figure 45 below.

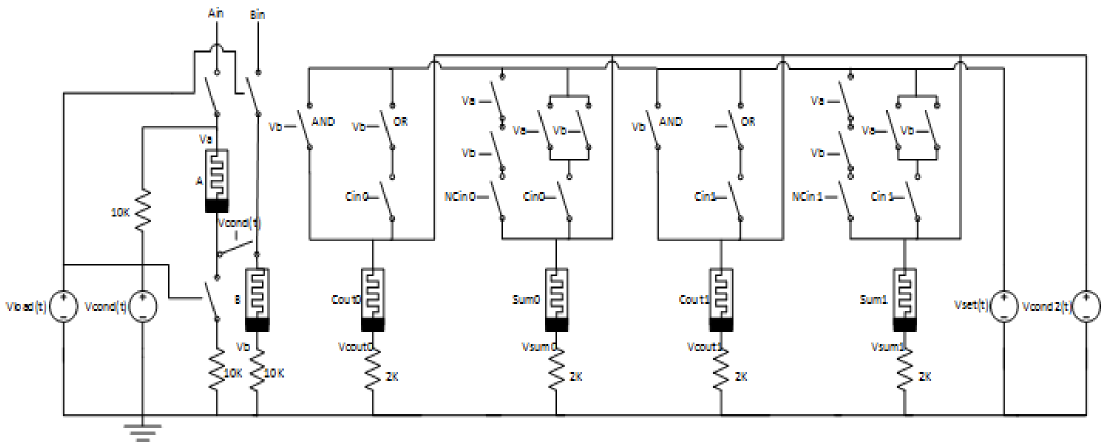


Figure 45: Optimized MAD adder for carry select adders

By replicating this adder k times and propagating both of the carry signals as described in prior work each k-bit block will require k steps to produce the final two carry-out values. The propagation of the carry-bit between blocks will use the same logic as used internal to the k-bit blocks, requiring only a single delay. Thus the total delay for an N-bit MAD carry select adder is (k)+(N/k-2)+1 or k+N/k-1. Solving the derivative for k results in the same optimal value of k as in CMOS, √(N). Thus, the total delay of an n-bit MAD carry select adder is 2√(N)-1, less than half the CMOS delay of 4√(N)+2.

A waveform of a 16-bit MAD carry select adder completing in 7 steps is shown in Figure 46.

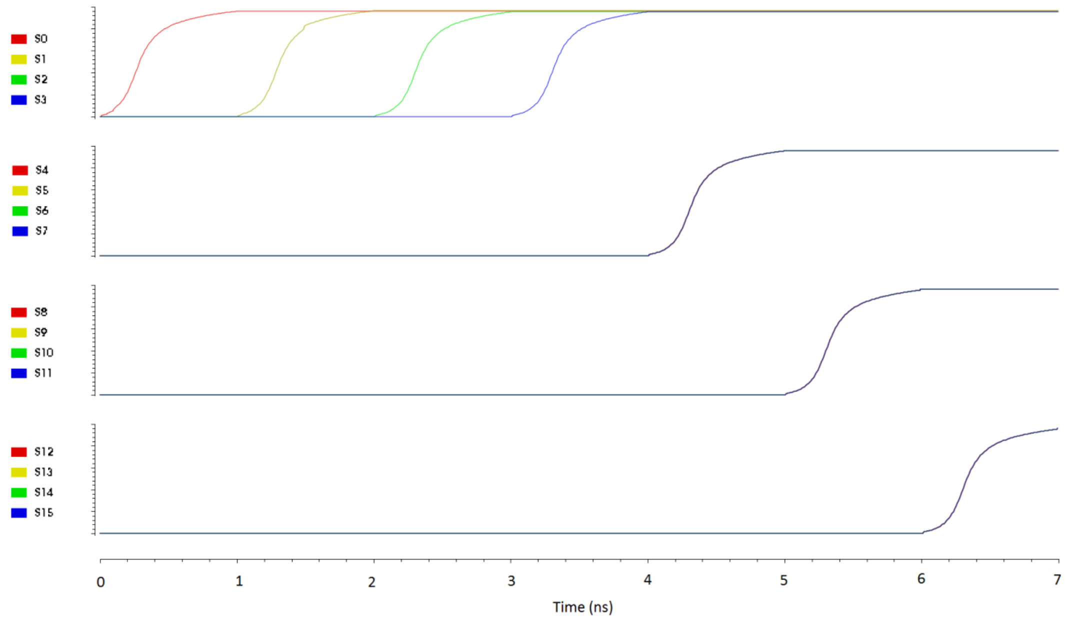


Figure 46: 16-bit MAD carry select addition for 0x0 + 0xFFFF

The first k bits resolve in the first k steps. Then, each k-bit block resolves in each subsequent step - Bits 4-7 resolve in step 5, bits 8-11 in step 6, and bits 12-15 in step 7. Similar to the ripple carry adder, the carry select adder circuitry can be pipelined to begin a new addition every 4 steps.

The area of the adder is also less than alternative designs. Since each k-bit block only requires a single k-bit ripple carry adder, the componential area is 6k memristors, 7k resistors, 22k switches, and 3k+1 drivers per block. The first k-bit block is also simplified to a standard MAD ripple carry adder since it has a known carry-in value. The carry propagation logic requires 1 memristor, 1 resistor, 3 switches, and 1 driver in order to select the carry-out signal from the two candidates and store its value. The circuitry for this is shown in Figure 47.

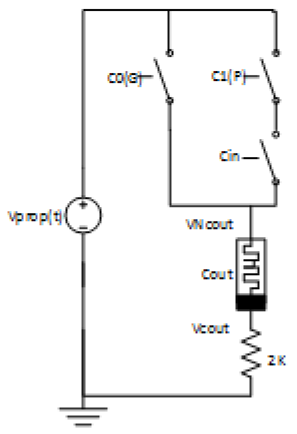


Figure 47: Carry propagation logic for the MAD carry select adder

V_{prop} applies a V_{set} signal to the carry-out memristor which is gated by the equation G+PC. The functionality of this equation is achieved in the same manner as is done for the full adder, taking 1 cycle. Therefore, the total componential area for an N-bit carry select adder is (N-k)(6 memristors + 7 resistors + 22 switches + (3+1/k) drivers)+k(4 memristors + 5 resistors + 13 switches + 3+1/k drivers) for the adders and ((N/k)-2)(1 memristor + 1 resistor + 3 switches + 1 driver) for the carry propagation logic for a total of [6N+N/k-2] memristors + [7N+N/k-2] resistors + [22N+3N/k-6] switches + [3N+2N/k-2] drivers. A schematic for a 16-bit MAD carry select adder is shown in Figure 48. Redundant drivers are replicated in the schematic for readability but many can be shared and optimized out as described.

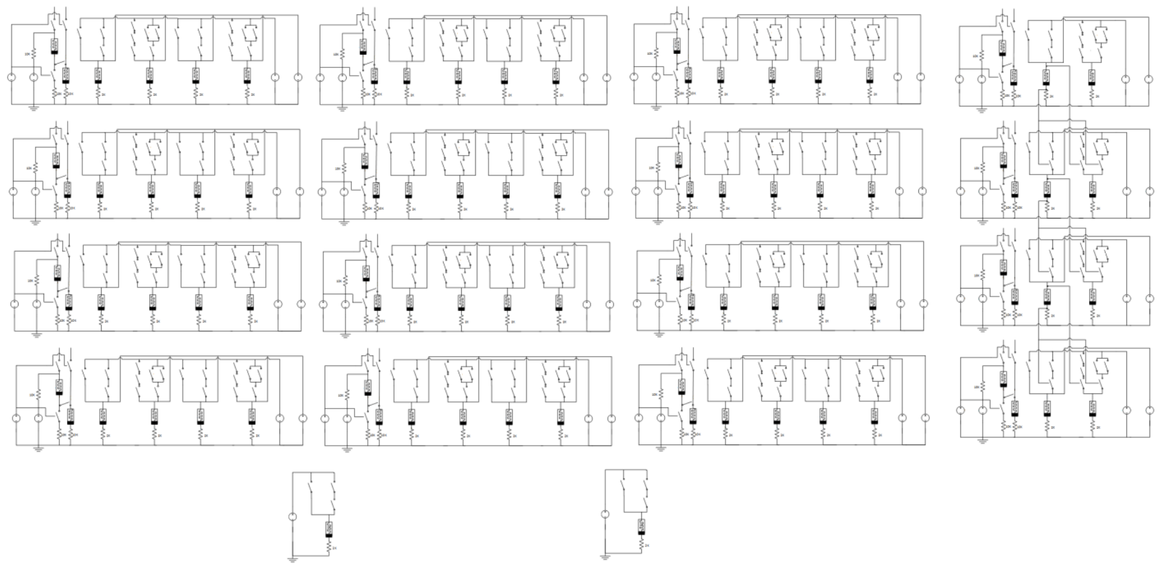


Figure 48 – 16-bit MAD carry select adder

CARRY SELECT ADDER ANALYSIS AND COMPARISON

A comparison of the delay of the proposed MAD design as compared to alternative approaches in prior work are shown in Table 13.

Table 13: Delay comparisons for the proposed MAD carry select adder

Approach	**Delay**	**Delay for k=sqrt(N)**
IMPLY	2k + 14 + 2(N/k) IMPLY	4√(N)+14
Hybrid	2k - 3 + 4(N/k)	6√(N)-3
GOTO Threshold Gates	k+N/k+1 GOTO	2√(N) + 1
Proposed MAD	2k-1	2√ N)-1
CMOS	2k+2N/k+2	4√(N)+2

In comparison to the traditional CMOS approach, the IMPLY approach maintains the same order critical path. The hybrid-CMOS design has an increased delay due to signal buffering on the carry propagation signals. The threshold gate and proposed MAD designs exhibit half the delay of traditional CMOS with the MAD design edging out the threshold design by a constant 2 delays. The delay is also less than half of the delay of the IMPLY design and about 1/3 of the Hybrid-CMOS design's delay.

Recall that both the IMPLY and MAD designs can also be pipelined to allow for higher throughput. Table 14 shows the latency for each of the proposed approaches to complete X consecutive additions

Table 14: Latency for X additions on the proposed carry select adders

	CMOS	IMPLY	Hybrid-CMOS	Threshold	MAD
Delay	$4X\sqrt{(N)}+2X$	$4\sqrt{(N)}+22X-8$	$6X\sqrt{(N)}-3X$	$2X\sqrt{(N)}+X$	$2\sqrt{(N)}+4X-5$

When multiple additions are considered, the benefits of the MAD design are magnified. The delay reduction as compared to the CMOS and hybrid-CMOS approaches is now multiplicative. The threshold approach, which appeared comparable for a single addition, is less competitive since no pipelining can be performed. As the number of additions increases, the threshold approach requires $(2\sqrt{(N)})X$ whereas the MAD approach requires a constant 4X. Thus, for all adder widths greater than 4, the MAD approach has a lower latency than the threshold gate approach. The IMPLY approach can also be pipelined to begin a new addition every 22 steps. Although this is not as performative as the MAD approach, it improves the latency to a constant 22X, rendering it a viable alternative over the other three approaches.

A comparison of the complexity of the proposed MAD design as compared to alternative approaches in prior work are shown in Table 15.

Table 15: Component counts for carry select adders

Approach	Memristors	Transistors	Resistors	GOTOs	Drivers	Switches
IMPLY	17N – 10k + 3N/k – 3		3N-2k+N/k-1		39N – 24k + 7N/k – 7	22N – 14k + 4N/k – 4
Hybrid-CMOS	34N-20k+4N/k-8	32N-16k+2N/k-2				
GOTO Threshold Gates	19N-12k+4N/k-7			6N-4k+N/k-2		
Proposed MAD	6N+N/k-2k-2		7N+N/k-2k-2		3N+2N/k-2	22N+3N/k-9k-6
CMOS		93N-54k+12N/k-22				

In comparison to the traditional CMOS design, the alternatives proposed in this work all require fewer components. The number of transistors is reduced by about 66% for the hybrid-CMOS approach. The other proposed designs do not require any transistors. However, the MAD and IMPLY approaches do require a non-negligible number of switches and drivers which are not required in the CMOS, hybrid-CMOS, or threshold designs. The total number of components is lesser for each of the proposed designs as compared to the CMOS design.

The proposed MAD design uses significantly fewer components that the CMOS and hybrid-CMOS designs. The IMPLY approach, often chosen for its low area, has a significantly higher area. Although the number of switches is roughly equivalent between the two designs, the IMPLY approach uses about 10 times the number of drivers and 3 times the number of memristors. The IMPLY approach does use around half the number of resistors but these components are generally negligible in designs. The

GOTO threshold gate approach requires about the same number of components as the MAD design, however the dimensional size comparison is left for future work.

Conditional Sum Adders

Memristor-based conditional sum adders are presented for the IMPLY, hybrid-CMOS, threshold, and MAD contexts.

IMPLY Conditional Sum Adder

A memristor-based conditional sum adder designed from IMPLY-based modified half adders and multiplexers is proposed, simulated, and analyzed. A breakdown of the execution steps to implement a 1-bit modified half adder with IMPLY operations is given in Table 16.

Table 16: Steps for a 1-bit IMPLY modified half adder

Step	**Functionality**	**Description**
1	A imp 0 -> S1	NOT A
2	B imp 0 -> M0	NOT B
3	M0 imp 0 -> C1	copy of B
4	A imp M0 -> M0	A imp (B imp 0) = A NAND B
5	S1 imp C1 -> $\mathbf{C_1}$	(A imp 0) imp B = A OR B = $\mathbf{C_1}$
6	M0 imp C0 -> $\mathbf{C_0}$	A AND B = $\mathbf{C_0}$
7	False M0,S1	
8	C1 imp S1 -> S1	NOT C1
9	C0 imp M0 -> M0	NOT C0
10	M0 imp S1 -> $\mathbf{S_1}$	(NOT C0) imp (NOT C1) = (NOT C0) NAND C1 = $\mathbf{S_1}$
11	S1 imp S0 -> $\mathbf{S_0}$	(NOT C0) AND C1 = $\mathbf{S_0}$

The execution takes a total of 11 total steps, 5 and 6 for the carry signals and 10 and 11 for the sum bits. This design intentionally resolves C_1 and S_1 before the C_0 and S_0 counterparts to hide latency in the adder design. These signals feed multiplexers such that C_0 and S_0 are paired with the inverse of the select line. Since the select line must be inverted before it can be ANDed with the operand, C_0 and S_0 can resolve a cycle later than C_1 and S_1 without an increase in latency. Note that the sum computations are delayed if the carry out values are set and sensed for propagation after resolution. A single bit modified half adder requires 2 input memristors, 1 working memristor, and 4 output memristors for C_0, C_1, S_0, and S_1 for a total of 7 memristors. The schematic is shown in Figure 49.

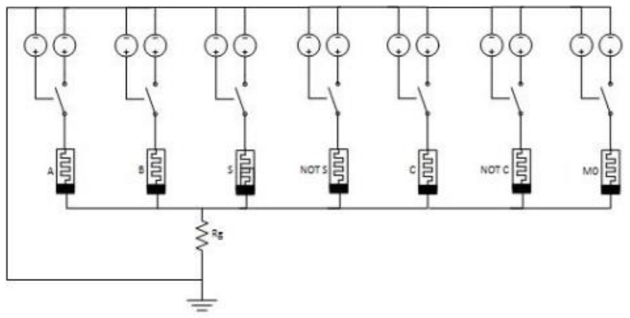


Figure 49: IMPLY modified half adder

The functionality of this circuit for inputs A=1 and B=0 can be seen in Figure 50.

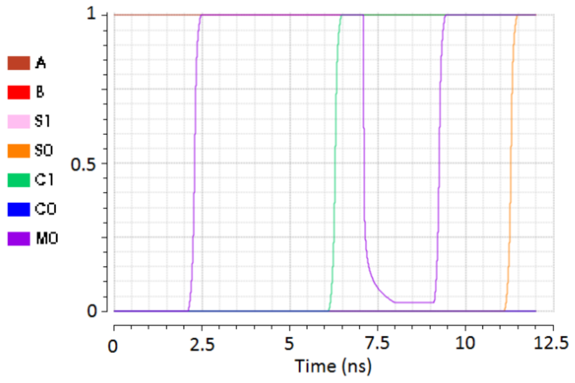


Figure 50: IMPLY modified half adder execution for inputs A=1 and B=0

For a conditional sum adder, each of the outputs from the MHAs and the select lines are set and sensed before the multiplexers because the fanout is greater than 1. The only exception is the first two bits which have a fanout of 1. However, some cycles can be saved by leveraging the fact that S_1 == NOT S_0. Referring back to Table 16, step 11 is essentially the "set" stage for S_1. This step can be expanded to S1 imp 0 $\rightarrow X_0,X_1,\ldots,X_N$ for the N multiplexers that require S_0. Then a single additional sense stage can perform

X_i imp 0 → Y_i for each X memristor in step 11. Thus, in 12 stages a modified half adder can compute all four outputs and set and sense both sum output bits. The same logic cannot be applied to C_0 and C_1 as they are not inverses of each other.

It was shown that the delay through a multiplexer is 4 delays + 2 for setting and sensing for a total of 6 delays. These multiplexers are used in a hierarchical fashion with the modified half adders to propagate the carry signal efficiently. A schematic for an 8-bit conditional sum adder based on IMPLY operations is shown in Figure 51.

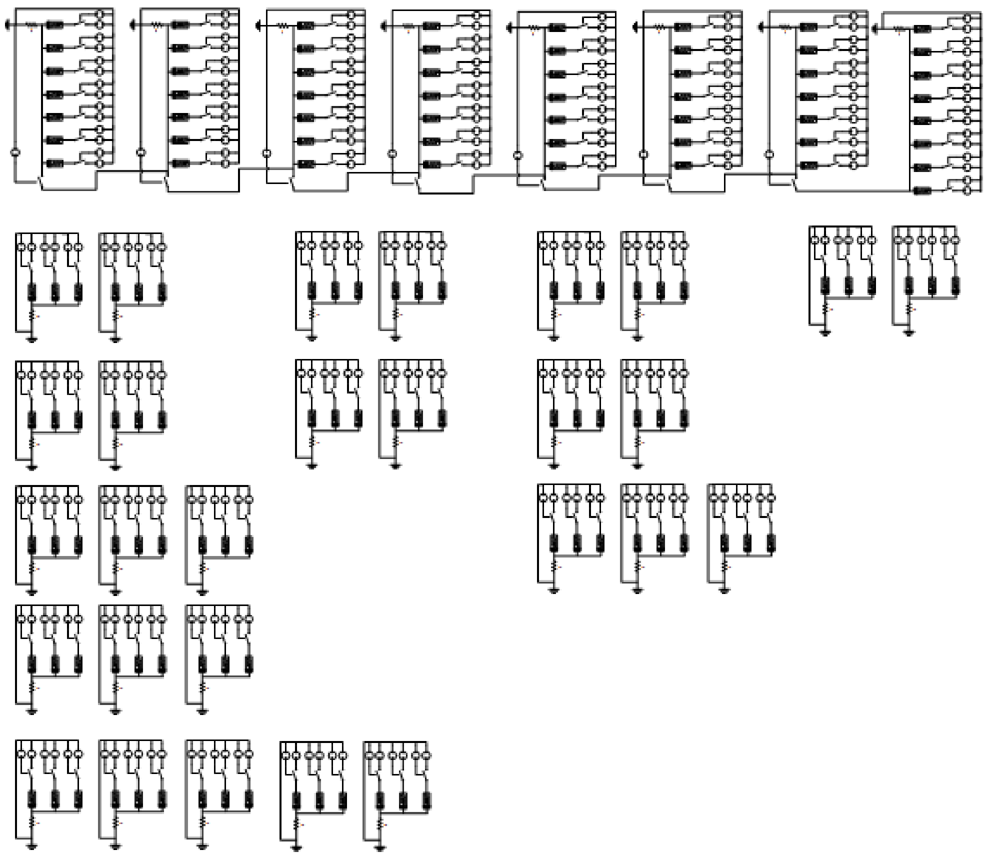


Figure 51: 8-bit IMPLY conditional sum adder

The modified half adders can be seen across the top row of the schematic, with the corresponding 1-bit multiplexers under the corresponding MHAs. The total complexity is 141 memristors. A summary of the complexity for various adder widths is given in Table 17.

Table 17: Complexity for IMPLY conditional sum adders

Width	**2**	**4**	**8**	**16**
Prior [15]	Not Given	Not Given	454 memristors	975 memristors
Proposed	21 memristors	56 memristors	141 memristors	332 memristors

This design has been further optimized for delay. In this design and the traditional CMOS design, the first bit is resolved with a full adder. However, a modified half adder with a 2-bit multiplexer can be used instead as done for other bits. A schematic for such an approach is shown in Figure 52.

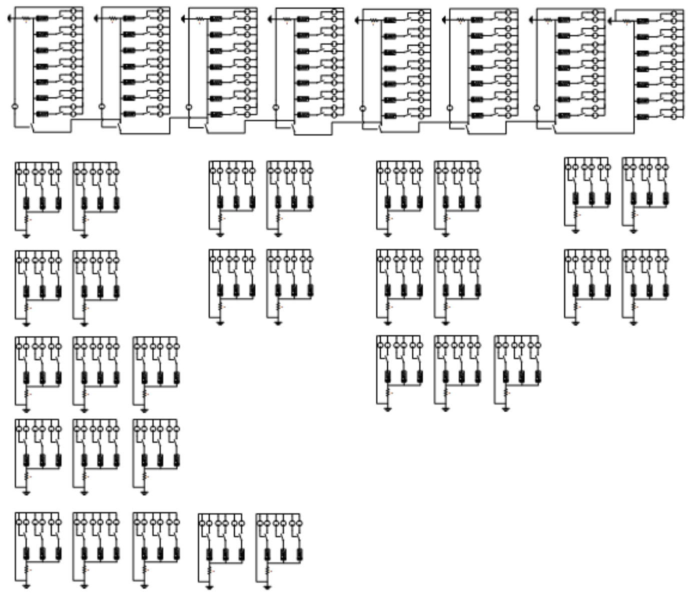


Figure 52: Optimized IMPLY conditional sum adder using only modified half adders and multiplexers

This design requires 5 more memristors but improves the delay by 3. The final carry out bit is resolved at step 27 rather than 30. This is one example where the design tradeoffs made for the CMOS design do not match those made for the IMPLYK design.

HYBRID-CMOS CONDITIONAL SUM ADDER

A hybrid-CMOS memristor-based conditional sum adder is presented and analyzed. In the hybrid-CMOS approach, a modified half adder consists of 6 memristors, and 4 MOSFETs as shown in Figure 53.

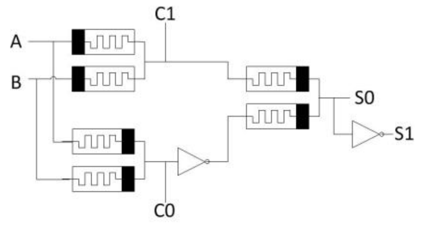


Figure 53: Hybrid-CMOS modified half adder

The carry out bits are ready after a single gate delay and the sum bits are ready after 3 and 4. A waveform for each possible input of A and B is shown in Figure 54.

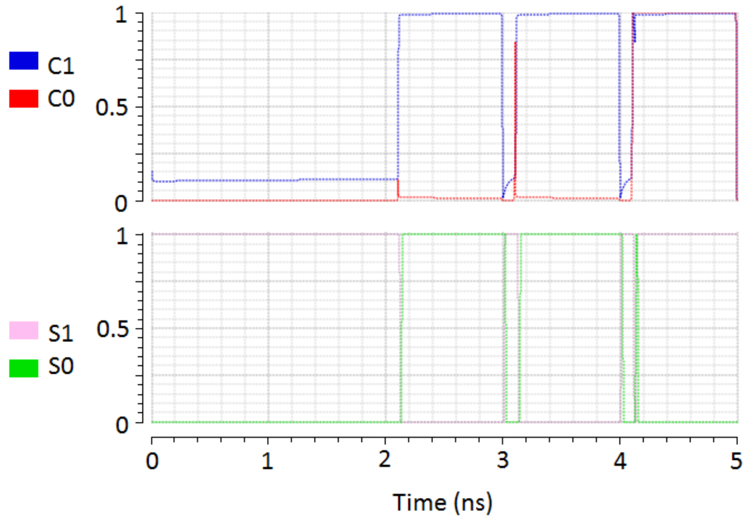


Figure 54 – Hybrid-CMOS modified half adder execution for all inputs

A schematic for an 8-bit conditional sum adder is given in Appendix C. The delay and complexity for hybrid-CMOS multiplexers is the same as CMOS multiplexers. The component count for a hybrid-CMOS multiplexer is 2 MOSFETs and 6 memristors. At the gate level, the complexity is nearly identical to the traditional CMOS counterpart except for the additional buffering required for the hybrid-CMOS implementation. The gates themselves in the hybrid-CMOS design are smaller. The specific component requirements for various adder sizes are given in the analysis section.

THRESHOLD GATE CONDITIONAL SUM ADDER

Conditional sum adders can also be implemented using threshold gates. For the purposes of conditional sum adders, it is preferential to use the CSTG implementation because it produces the result of the threshold detection and its inverse whereas the GOTO implementation only produces the threshold detection result. This fact allows a modified half adder to be created from threshold gates as shown in Figure 55.

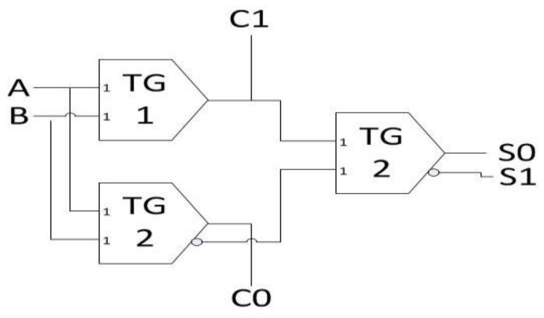


Figure 55: CSTG threshold gates modified half adder

This modified half adder has a complexity of 3 threshold gates or 6 memristors, 5 resistors, 24 transistors, and 9 vias. The delay is 1 for the carry out bits C_0 and C_1 and 2 for the sum out bits S_0 and S_1. Recall an N-bit multiplexer with CSTG threshold gates has a delay of 2 and complexity 2N+1 gates for a total of 18N+2 transistors, 2 vias, 5N memristors, and 5N resistors.

For a conditional sum adder which uses a modified half adder for its first bit as well as the remaining bits, the total delay is 2k+2 where $k=1+\log_2(N)$. This is an improvement over the other proposed implementations and the standard CMOS implementation.

MAD Gate Conditional Sum Adder

This section presents the design of an 8-bit MAD-based conditional sum adder architecture. As described for the MAD carry select adders, each of the modified half adders can produce its four outputs, $C0_i$, $C1_i$, $S0_i$, and $S1_i$ in a single step once the carry-in is available. The circuit for this modified half adder is shown in Figure 56.

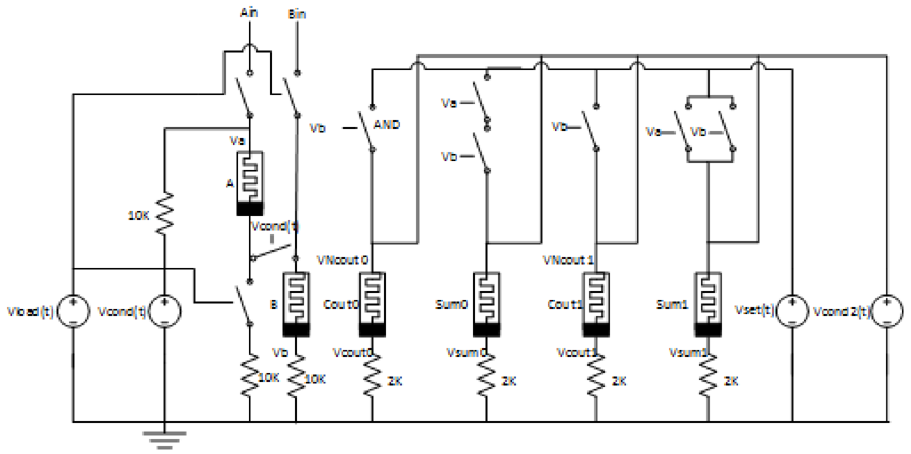


Figure 56: MAD modified half adder for even bits in a conditional sum adder

A naïve approach would use this modified half adder design for every bit of the adder. Then, the first level 2-bit muxes would require an additional 8 switches and 2 memristors per multiplexer. However, this incurs a cost in both area and delay that is unnecessary if the design is optimized for the adder's context. Let the even bits' modified half adders resolve the four outputs in step 1 using the circuit in Figure 56. In the next step, both the modified half adders for the odd bits and the multiplexers can resolve. The modified half adders for the even bits apply V_{cond2} to the carry-out memristors. The voltages at Vc_{out0} and Vc_{out1} are then sensed for their values and used in conjunction with the logic in the next bit as shown in Figure 57.

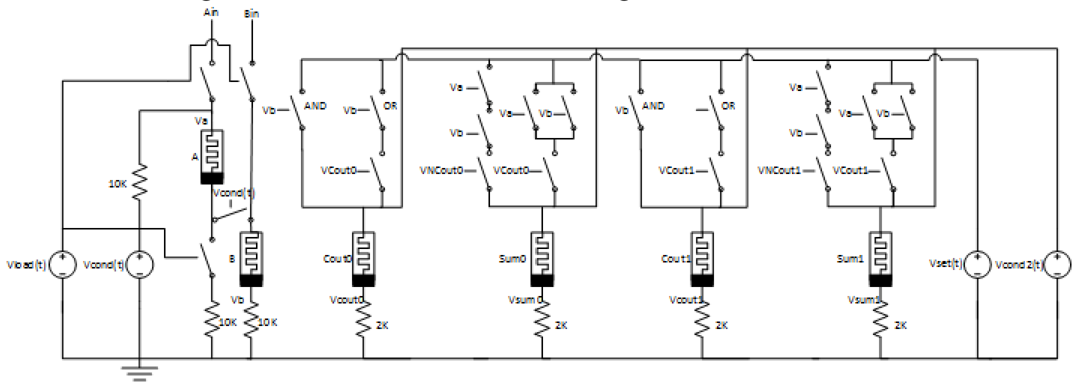


Figure 57: MAD modified half adder for odd bits in a conditional sum adder

This removes the need for the multiplexer logic by incorporating it into the modified half adders. The values of the output memristors can now be sensed in the same manner as described for the even bits and propagated to multiplexers in later levels of the addition. Thus, each level of the conditional sum adder incurs a single bit delay. The total delay for an N-bit conditional sum adder is log(N)+1.

All of the remaining multiplexers require 1 memristor, 1 resistor, 4 switches, and 2 drivers per output each. The circuitry for these multiplexers is shown in Figure 58.

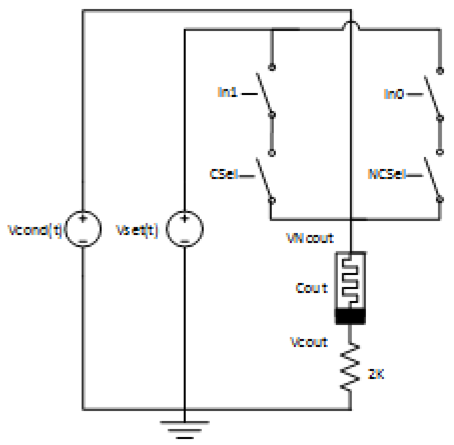


Figure 58: MAD multiplexer for a conditional sum adder

The V_{set} signal is applied at the same time that the V_{cond} signal is applied to both the inputs of the multiplexer, denoted by In1 and In0, and to the carry-out signals used as the select line. Whenever this output is needed, the V_{cond} signal is applied to facilitate reading its voltage. The V_{set} and V_{cond} signals can be shared across all multiplexers on the same level in the adder design as long as these values are set and read at the same steps in execution.

The total complexity consists of the area of the modified half adders and the multiplexers. A schematic for an 8-bit MAD conditional sum adder is shown in Figure 59.

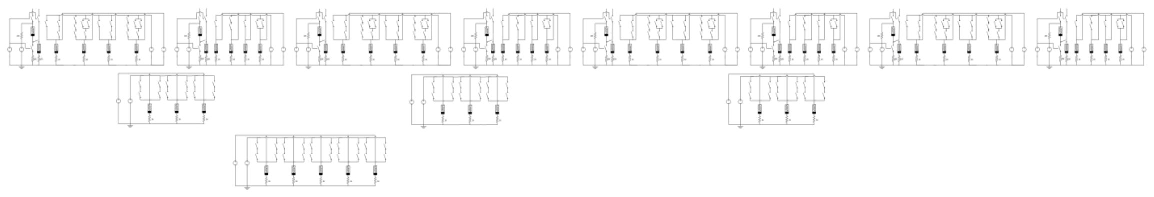


Figure 59: 8-bit MAD conditional sum adder

Redundant drivers are shown to make functionality clear however many can be optimized out. For example, as described earlier, all of the multiplexers on the same level in the design can share the V_{cond} and V_{set} drivers. By the same logic, all of the even-bit adders can share their drivers and all of the odd-bit adders can too. This implies that the number of drivers only increases by 2 each time the adder width doubles. Even-bit modified half adders require a total of 3N memristors, 7N/2 resistors, and 5N switches. The odd bit modified half adders require a total of 3N memristors, 7N/2 resistors, and 11N switches. Each 1-bit multiplexer requires 1 memristor, 1 resistor, and 4 switches. Thus, for an 8-bit addition the total complexity is 62 memristors + 70 resistors + 184 switches + 12 drivers.

The MAD design requires log(N)+1 steps to complete an addition. In other words, the delay only increases by a single step each time the width of the adder doubles. A waveform showing the completion of an 8-bit addition in 4 steps is shown in Figure 60.

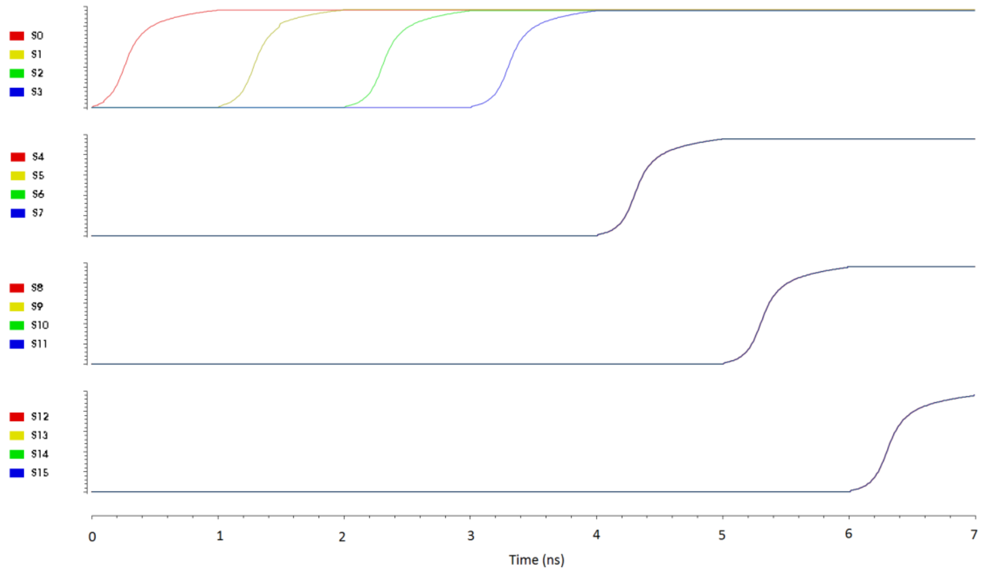


Figure 60: 8-bit MAD conditional sum adder execution for 0x0 + 0xFF

The first sum bit resolves in the first step and the second bit resolves in the second. After that, each level of multiplexers resolves in a subsequent step.

The first level of the adder is complete after 2 steps. Thus, to pipeline additions, the circuitry can reset its memristors in step 3, load the inputs in step 4, and begin execution again in step 5. Thus, similar to the MAD ripple carry adder and carry select adder, a new addition can begin every 4 steps.

CONDITIONAL SUM ADDER ANALYSIS AND COMPARISON

A full componential comparison for the proposed MAD conditional sum adder and prior work for a range of adder widths is shown in Table 18.

Table 18: Complexity for conditional sum adders

Adder Width	**2**	**4**	**8**	**16**
CMOS	90 MOSFETs	239 MOSFETs	614 MOSFETs	1471 MOSFETs
IMPLY	21 memristors 4 resistors 21 drivers 25 switches	56 memristors 13 resistors 56 drivers 69 switches	141 memristors 38 resistors 141 drivers 179 switches	332 memristors 89 resistors 332 drivers 421 switches
Hybrid-CMOS	36 memristors 12 MOSFETs	90 memristors 26 MOSFETs	228 memristors 56 MOSFETs	546 memristors 118 MOSFETs
Threshold CSTG	62 MOSFETs 16 memristors 16 resistors	180 MOSFETs 47 memristors 47 resistors	528 MOSFETs 134 memristors 134 resistors	1352 MOSFETs 343 memristors 343 resistors
MAD	12 memristors 14 resistors 8 drivers 32 switches	27 memristors 31 resistors 10 drivers 76 switches	62 memristors 70 resistors 12 drivers 184 switches	139 memristors 155 resistors 14 drivers 428 switches

The IMPLY implementation does not require any CMOS components and its component counts are consistently less than the CMOS for the 16-bit adder. The hybrid-CMOS approach uses more memristors than the IMPLY approach and requires additional MOSFETs for the NOT gates. This complexity is still less than that of traditional CMOS by a significant margin. The reduction in components in the hybrid-CMOS implementation increases as the adder size increases. For the 16-bit adder, the hybrid-CMOS implementation uses less than 10% as many MOSFETs as the CMOS implementation. The threshold gate implementation reduces the number of MOSFETs slightly as compared to CMOS, however the memristors and resistors are added to the design such that overall component count increases. .

The MAD design uses fewer components than all alternative designs. Also, the MAD design uses no MOSFETs and less than half the number of memristors as compared to the threshold and hybrid-CMOS designs for adder widths of 8 and greater. The IMPLY approach, known for its low area, requires about 4 times as many memristors, 50% more switches, and almost 100x the number of drivers as the adder size increases. For the 16-bit designs, the IMPLY design requires 1187 drivers while the MAD design requires only 12. This renders the MAD design a very low area design.

Delay comparisons for the proposed conditional sum adders are shown in Table 19.

Table 19: Delay for conditional sum adders, $k = 1+\log_2(N)$

Adder Width	**2**	**4**	**8**	**16**	**General**
CMOS	8	11	14	17	3k+2
IMPLY	12	17	22	27	5k+6
Hybrid	8	11	14	17	3k+2
Threshold CSTG	6	8	10	12	2k+2
MAD	2	3	4	5	k

The hybrid-CMOS approach has an equivalent delay to the traditional CMOS implementation as expected. The MAD design has the lowest delay, with a 32-bit addition taking only 6 cycles. This design has less than half of the delay of the next best alternative, threshold gates, which require 2log(N)+4 steps. The remaining alternatives, CMOS, hybrid-CMOS, and IMPLY, all take at least three times as long to complete. The benefits of the MAD approach grow as the adder width increases since each time the adder width doubles, the design only requires 1 additional step. The IMPLY approach requires 5, the threshold requires 2, and the hybrid-CMOS and CMOS approaches require 3.

The IMPLY and MAD approaches can again be pipelined, allowing a new addition to begin every 23 and 4 steps respectively. Thus, for large numbers of consecutive additions, the IMPLY approach becomes a better alternative than the threshold, hybrid-CMOS, or CMOS approaches, and the MAD approach gains an even larger delay advantage over all other approaches.

Carry Lookahead Adders[4]

Memristor-based carry lookahead adders are presented for the IMPLY, hybrid-CMOS, threshold, and MAD contexts.

IMPLY Carry Lookahead Adder

The IMPLY implementation for a 16-bit carry lookahead adder \reduces the area and the delay for the final carry-out bit and sum bit significantly as compared to prior work by recognizing Boolean-based simplifications in the IMPLY domain. In carry lookahead adders, the AND operation is often followed by the OR operation or a second AND operation. By treating these operations as single operational units, the IMPLY operation can be simplified as shown in Table 20.

Table 20: Optimized Boolean equations using the IMPLY operation

Operation	**Baseline**	**Optimized**
AB+C	1. B imp 0 2.A imp (B imp 0) 3.(A imp (B imp 0)) imp 0 = AB 4.AB imp 0 5.C imp 0 6.(C imp 0) imp 0 = Copy of C 7.(AB imp 0) imp C = AB+C	1.B imp 0 2.A imp (B imp 0) 3.(A imp (B imp 0) imp C = AB +C
AB*C	1.B imp 0 2.A imp (B imp 0) 3.(A imp (B imp 0)) imp 0 = AB 4.C imp 0 5.AB imp (C imp 0) 6.(AB imp (C imp 0) imp 0 = AB*C	1.B imp 0 2.A imp (B imp 0) 3.C imp (A imp (B imp 0)) 4.(C imp (A imp (B imp 0)) imp 0 = AB*C

If the IMPLY operations are serialized, an AND operation followed by an OR operation requires 7 steps and 8 memristors and two consecutive AND operations

[4] L. Guckert and E. E. Swartzlander Jr., “Carry Lookahead Adder Design Using Memristors,” Chapter in *Devices, Circuits and Systems*, CRC Press, Oct. 2016. This section is an extension of previously published work for which Lauren Guckert is the principal author and responsible for all content.

requires 6 steps and 8 memristors. These are shown as the baseline approach in Table 20. Note that steps 3 and 4 for the AB+C operation are redundant. Two consecutive NOT operations are performed, which performs a copy operation in effect. This is done based on the assumption that the intermediate value of A AND B resolved in step 3 is needed. However, in the context of a carry lookahead adder, this result is only used to compute AB+C and AB*C. Thus, steps 3 and 4 and their two accompanying memristors can be removed. Similarly, the baseline approach assumes that the memristor C will be needed in subsequent operations. However, in a carry lookahead adder, this is not the case. Thus, steps 5 and 6 and their two memristors can also be removed. This results in just 3 steps and 4 memristors. Similar optimizations are performed to reduce the requirements for AB*C to 4 steps and 5 memristors.

Both of these Boolean expressions are used extensively in carry lookahead adders. The calculations for the group propagate and generate signals and the carry-out and sum values use chained instances of them. This leads to a multiplicative improvement in the overall delay and area of the design. The execution steps to calculate the generate (G), propagate (P), group generate (GG), and group propagate (GP) bits for a 4-bit block in a carry lookahead adder is given in Table 21.

Table 21: Steps for an optimized 4-bit block in an IMPLY carry lookahead adder

STEP	FUNCTIONALITY	GOAL
1	A imp 0	A->M0
2	B imp 0	B->M3
3	B	M3->M1
4	(A imp 0) imp (B imp 0)	M0->M3
5	A imp B	A->M1
6	((A imp 0) imp (B imp 0)) imp 0	M3->M2
7	A NAND B = **NOT G**	B->M0
8	A XOR B**= P**	M1->M2
9	False memristors	False M0,M1,M3
10	SET P	M2->M0
11	SENSE P / SET G	
12	SENSE G	
13	(P1 imp (G0 imp 0) etc.	NAND: P0C0 P1G0 P2G1 P3G2 P1P2 P0P1 P2P3
14	(P0 imp (C0 imp 0) imp G0=**C1** (P1 imp (G0 imp 0) imp G1 (P2 imp (G1 imp 0) imp G2 (P3 imp (G2 imp 0) imp G3 C0 imp (P0 imp (P1 imp 0)) G0 imp (P1 imp (P2 imp 0) G1 imp (P2 imp (P3 imp 0)) G0P1 imp 0 P0P1 imp 0 C0P0 imp 0	G0+P0C0 G1+P1G0 G2+ P2G1 G3+P3G2 NAND C0*P0P1 NAND G0*P1P2 NAND G1*P2P3 G0P1*P2P3 P0P1*P2P3 C0P0*P1P2
15	(C0P0 imp 0) imp P1P2 (P0P1 imp 0) imp P2P3 (G0P1 imp 0) imp P2P3 C0P1P0 imp (G1+P1G0) =**C2** G0P1P2 imp (G2+P2G1) G1P2P3 imp (G3+P3G2)	NAND C0P0*P1P2 NAND P0P1*P2P3 NAND G0P1*P2P3 C0P1P0 + (G1+P1G0) G0P1P2+(G2+P2G1) G1P2P3+(G3+P3G2)
16	((P0P1 imp 0) imp P2P3) imp 0 =**GP** (C0P0P1P2) imp ((G2+P2G1)+G0P1P2) =**C3** G0P1P2P3 imp (G1P2P3 + (G3+P3G2)) =**GG**	P0P1 AND P2P3 C0P0P1P2 + G0P1P2+(G2+P2G1) G0P1P2P3+ G1P2P3+(G3+P3G2)

All of the operations listed in a given step in the table can be performed in parallel. Also, each bit performs the first 12 steps in parallel. The first 8 steps focus on resolving the value of the individual bit's propagate and generate signals since these are first on the critical path. Next, the results of these values are copied as needed so that multiple copies exist for the remaining steps. By doing this, the remaining steps can all execute in parallel using local copies of the signals. Steps 13-16 are performed once per 4-bit block to resolve the internal carry signals as well as the group generate and group propagate signals.

In the traditional Boolean equations for a carry lookahead adder, nearly all of the outputs use the same intermediate expressions but in different steps. This implies that each step must be performed in a serialized manner or the values must be copied first to ensure values are not overwritten. However, both of these options result in a significant increase in delay and/or complexity. The proposed approach circumvents both of these options by translating the traditional Boolean expressions and eliminating the use of a single value across steps. The updated expressions are rewritten and ordered such that expressions sharing the same value(s) operate in parallel and all sense the value in the same step. Overall, this has the impact of reducing the delay from 38 to just 16. A waveform confirming the above behavior for inputs A=1111, B=0000, and C_{in}=1 is shown in Figure 61.

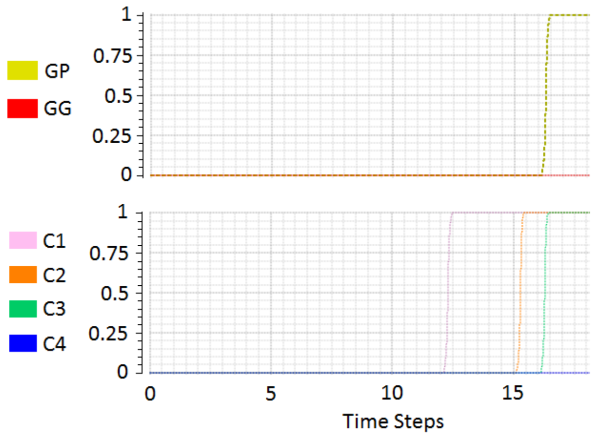


Figure 61: Resolution of the group generate (GG), group propagate (GP), and carry bits for a 4-bit IMPLY carry lookahead adder block

C1 is able to resolve 2 cycles earlier because its execution can begin in parallel with the set and sense states of G in cycles 11 and 12. GG and GP are resolved in cycle 16, C2 is resolved in cycle 15, and C3 is resolved in cycle 16. When each ith carry bit is resolved, the (i+1)th full adder resolves the sum bit as discussed in prior work [21]. A previous implementation [16] resolved the final carry-out bit at step 25 and final sum at 32 for a 4-bit adder. This work improves these delays to 16 and 20, a 33% improvement.

The carry lookahead adder delay grows logarithmically with the input. The delay from the GG and GP generation at level X to their generation at level X+1 is 5 delay cycles as compared to 4 cycles for traditional CMOS. A waveform for a 16-bit IMPLY carry lookahead adder for the resolution of the GG, GP, and C_i bits for inputs A=0xFFFF, B=0x0000, and C_{in}=1 is given in Figure 62.

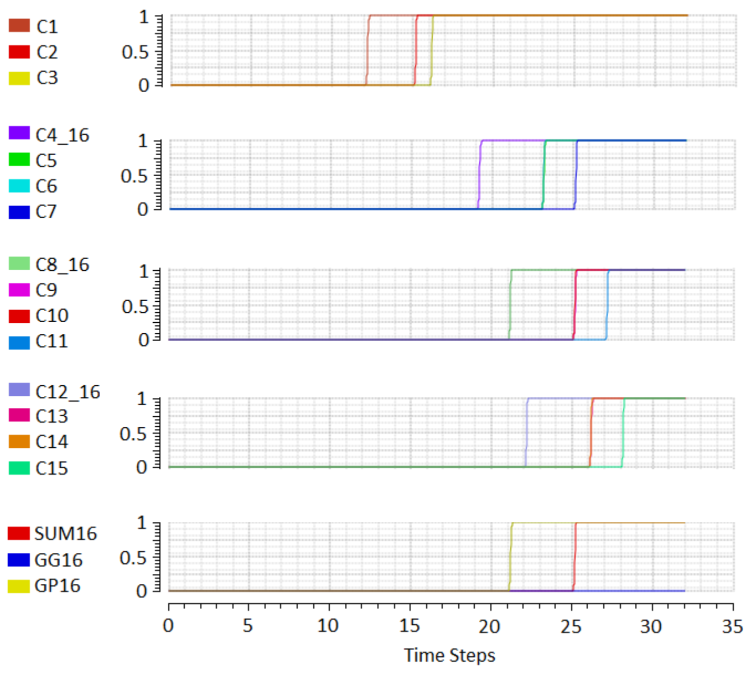


Figure 62: 16-bit IMPLY carry lookahead adder execution

C1, C2, and C3 are resolved as in Figure 61. Once GG and GP are resolved in the 4-bit blocks in cycle 16, the 16-bit block can begin resolving C4, C8, C12, and the 16-bit GG and GP bits. C4 is resolved at t=19n, C8 at t=21n, and C12 at t=22n. Then, each remaining 4-bit block can proceed in resolving its internal carry signals and sum bits. GP16 and GG16 are resolved at t=21n and finally the sum bits at t=26n. Note that the delay for the presented 16-bit adder is equivalent to the prior state-of-the art's delay for a 4-bit adder. This is indicative of the benefits one can expect when optimizing a design for the IMPLY context. This design can also be pipelined to allow a new addition to begin every 22 steps.

The design requires 7 memristors per bit for the calculation of P_i, G_i, and S_i and 49 memristors for the 4-bit group propagate, group generate, and C_i bits, for a total of 77 memristors and drivers per 4-bit block. Each group logic block also requires a switch. This logic can be replicated for the 16-bit group propagate, group generate, and C_i bits. However, memristors can be reused across the 4-bit and 16-bit blocks to minimize the total area by performing reset operations off the critical path. With this, the 16-bit block only requires 40 memristors for the group generate propagate and carry signals as compared to 49 in the 4-bit blocks. The total complexity of the 16-bit adder is 348 memristors and their drivers, 78 pull down resistors, and 370 switches. A full schematic of a 16-bit IMPLY based carry lookahead adder is shown in Figure 63.

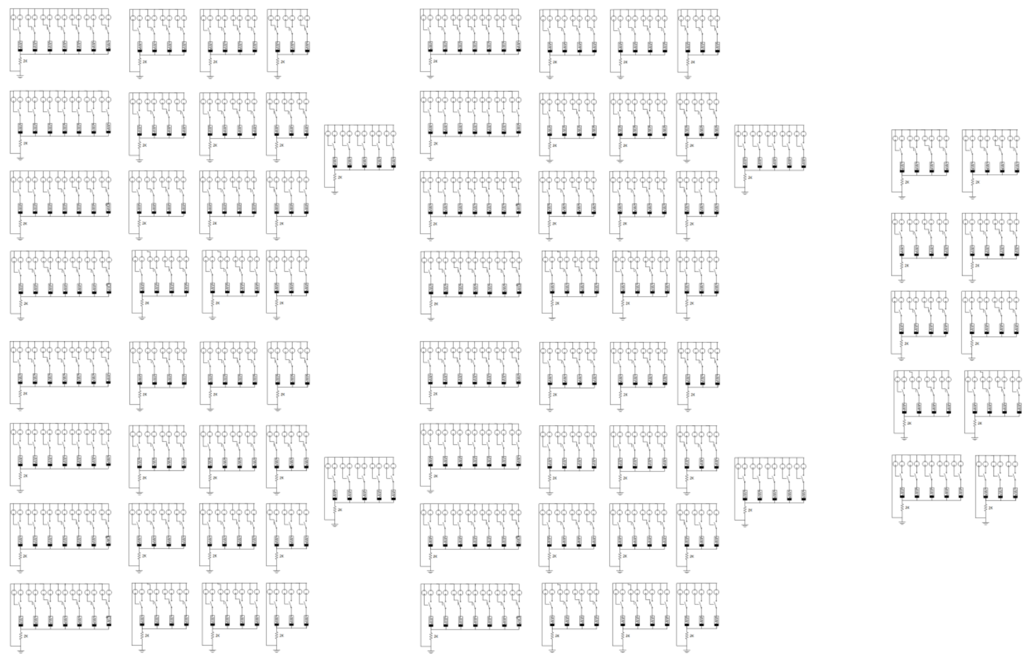


Figure 63: 16-bit IMPLY carry lookahead adder schematic

The four 4-bit blocks are shown in the symmetric four blocks on the left of the schematic. The left most blocks are the adders and the remaining memristors are for propagate, generate, and carry logic for each block. The far right logic is the propagate, generate, and carry logic for the 16-bit adder.

HYBRID-CMOS CARRY LOOKAHEAD ADDER

This section presents an optimized implementation of a 16-bit hybrid-CMOS carry lookahead adder. One consequence of hybrid-CMOS designs is their reliance on CMOS NOT gates. However, carry lookahead adders use very few instances of the NOT gate, since all of the generate and propagate logic (the majority of the logic in the overall design) can be implemented using only memristors. This observation makes carry lookahead adders a prime candidate for implementation in hybrid-CMOS. Using this knowledge and the presented optimized XOR gate, a 4-bit hybrid-CMOS adder block is proposed with a delay of 7 and 8 cycles for the carry-out and sum bits and a total complexity of 94 memristors and 40 MOSFETs. The schematic for this design is shown in Figure 64.

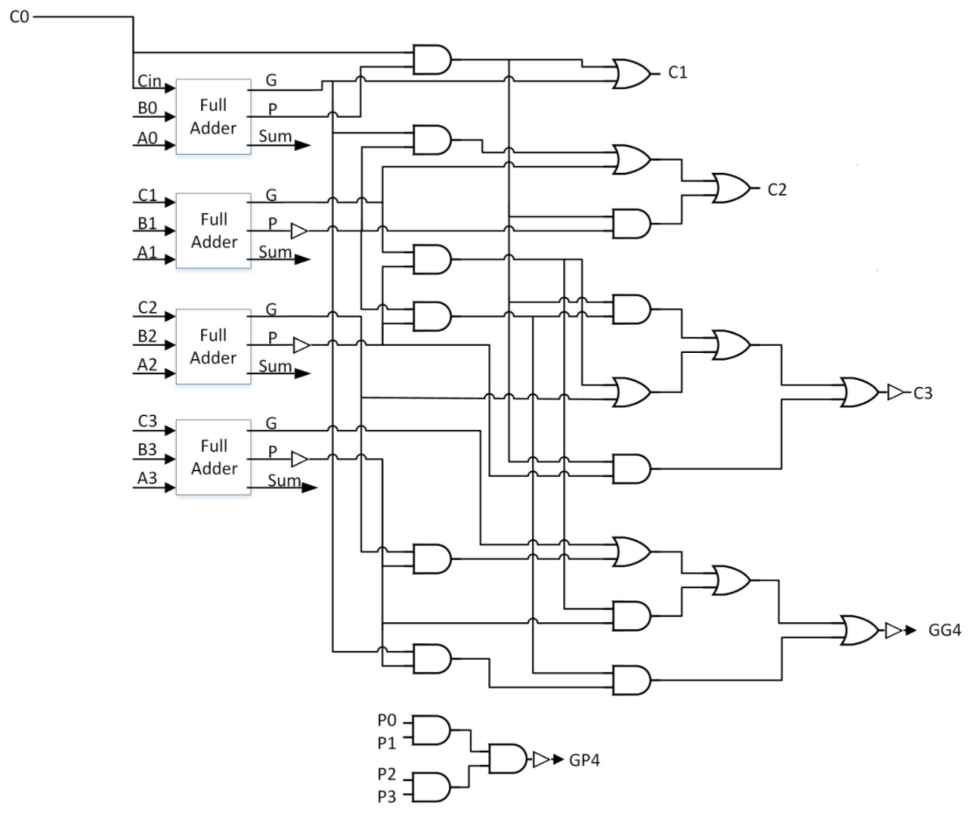


Figure 64: 4-bit Hybrid-CMOS carry lookahead adder schematic

In this case, the propagate signals from bits 1, 2, and 3, and C3 must be buffered. Additionally, the group propagate and group generate signals are buffered in preparation for their use in the larger adder. This causes the group propagate signal to be ready after 4 steps, the group generate and carry-out to be ready after 7, and the final sum to be ready after 8. The six buffers also contribute 24 MOSFETs to the design. Each adder requires 12 memristors and 4 MOSFETs to compute the generate, propagate, and sum signals. Thus, the total complexity of the 4-bit block is 94 memristors and 40 MOSFETs.

When the 4-bit blocks are extended to a 16-bit adder, additional signal restoration buffers must be added because both the fanout and the concatenation of gates increases for the outputs. This results in the schematic shown in Figure 65.

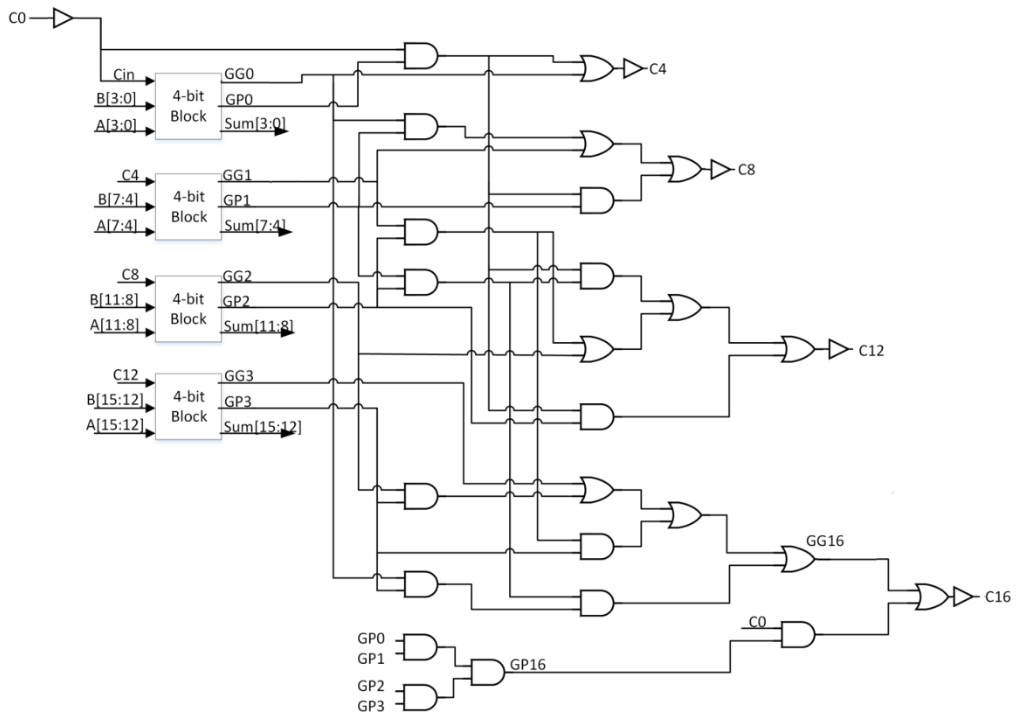


Figure 65: 16-bit Hybrid-CMOS carry lookahead adder schematic

The total complexity is 426 memristors and 176 MOSFETs. Note that buffers are not required on the outputs of the 4-bit blocks since these were included in the blocks themselves. The only necessary buffers are on the carry signals.

The delay for the carry bit and sum bit are 14 and 21 steps. The group generate signals from the 4-bit blocks are not resolved until step 7 and there are an additional 6 gate delays to resolve C12. As a result, C_{in} to the final block does not arrive until step 13. As shown for the 4-bit block, the delay for the sum is 8 bits so the final sum is resolved at step 21. All of the output signals and their resolution can be seen in Figure 66.

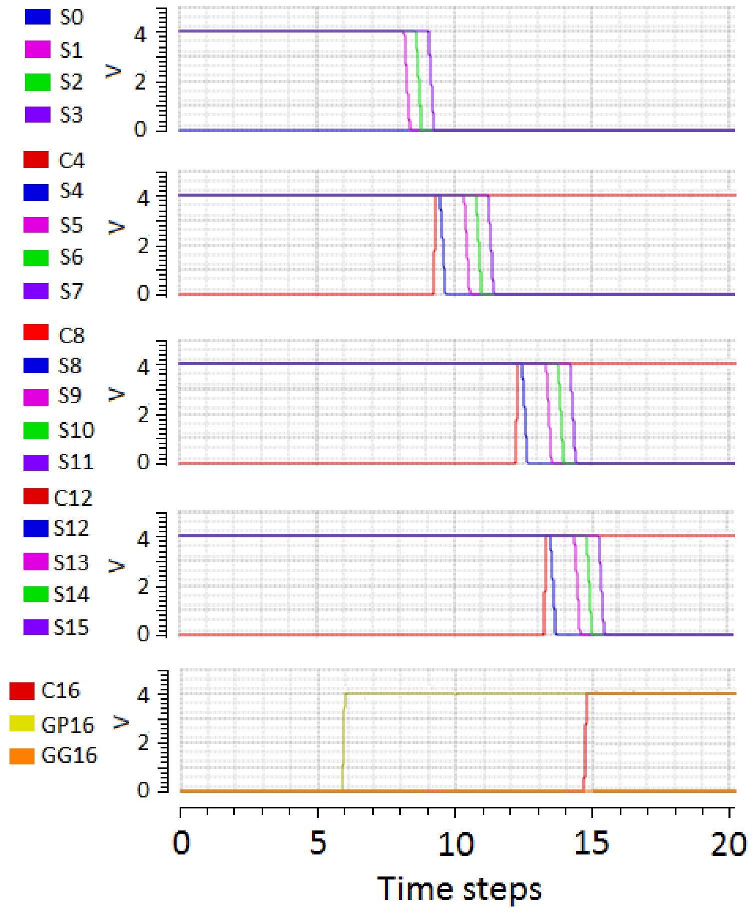


Figure 66: 16-bit Hybrid-CMOS carry lookahead adder execution

Each 4-bit block resolves the sum bits in consecutive cycles after the respective C_{in} arrives to the block. GP16 is resolved at step 6, two cycles after the group propagates from the 4-bit blocks are resolved, and GG16 at step 11.

THRESHOLD GATE CARRY LOOKAHEAD ADDER

A carry lookahead adder constructed from threshold gates achieves lower depth and lower gate count than alternative implementations. Threshold gates allow the traditional equations for propagate and generate signals to be rewritten to produce a smaller delay as shown in prior work [36]. A 4-bit carry lookahead adder can be constructed from 8 threshold gates and 4 NOT gates as shown in Figure 67.

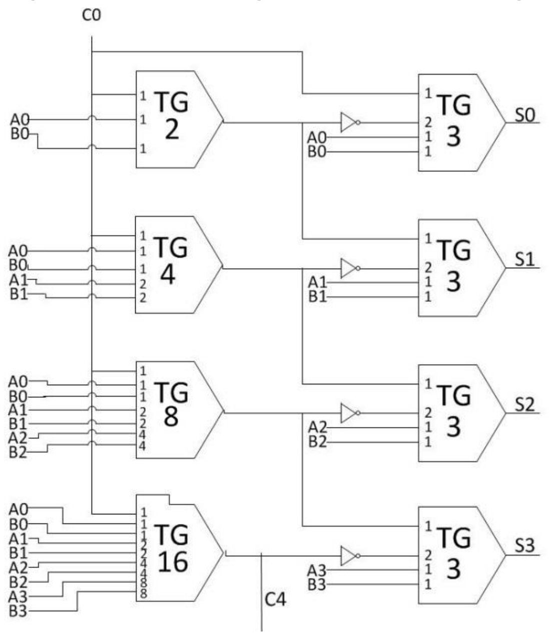


Figure 67: 4-bit threshold gate carry lookahead adder

It is not necessary to generate the generate or propagate signals for individual bits: Increased-input counts and varying input weights on the threshold gates achieve the same lookahead capabilities. This design reduces the carry-out delay to 1 and the sum delay to

3 due to the NOT gate. This design can be extended to a 16-bit carry lookahead adder in a hierarchical manner. Figure 68 shows a schematic for a 16-bit design.

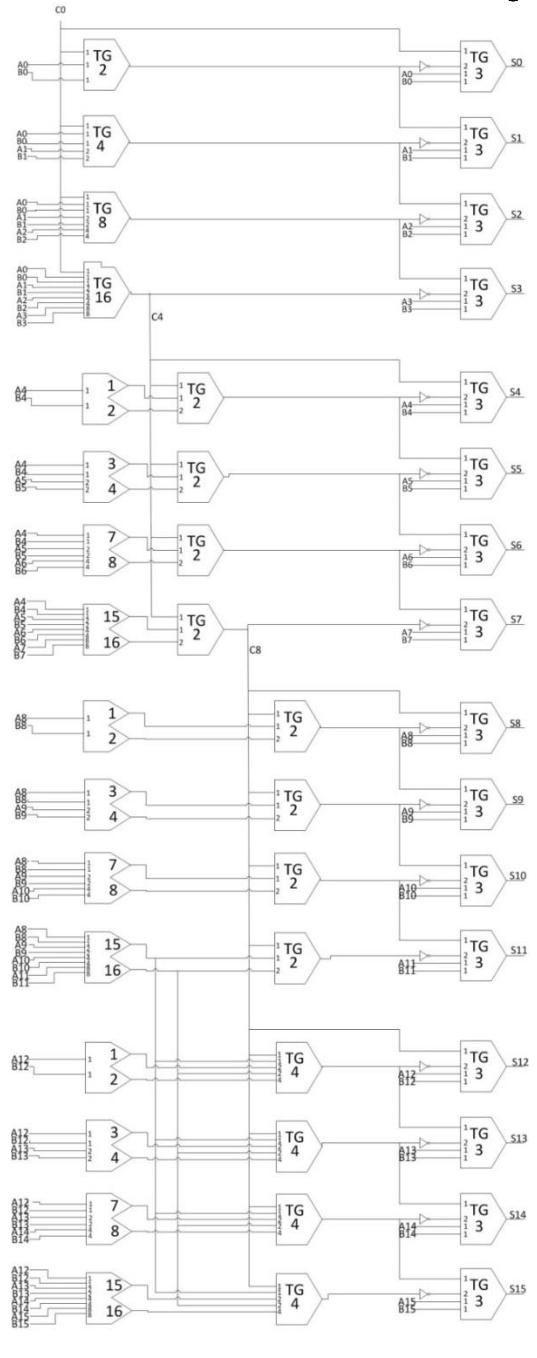


Figure 68: 16-bit threshold gate carry lookahead adder

In the first cycle, the 4-bit group generate and group propagate signals are resolved. In the second cycle, C_4 is used by the 2nd block to resolve the carry bits including carry out C_8. C_8 is then used in conjunction with the group propagate and generate bits of the final two blocks to generate the remaining carry out signals. In this manner, the final carry out bit C_{16} is resolved at cycle 3. Lastly, each of the carry signals is paired with the original input bits to resolve the sum bits in cycle 5.

This design reduces area by leveraging increased input counts on the threshold gates. Also, the 4-bit blocks can resolve their propagate and generate signals using different threshold detectors for the same inputs thus saving components. The entire design only requires 44 threshold gates and 32 MOSFETs.

MAD GATE CARRY LOOKAHEAD ADDER

This section presents a 16-bit carry lookahead adder constructed from MAD gates. First consider how to obtain P and G in the context of MAD designs. Propagate is equivalent to A OR B or A XOR B, and Generate is equivalent to A AND B. To obtain these Boolean values, A V_{cond} signal would be applied to the inputs A and B in series within each full adder while a V_{set} signal would be applied to an output memristor for G and for P. The voltage at the p terminal of the B memristor would be sensed to determine whether V_{set} was gated to the G or P memristor. Essentially, P = (V_b == OR) and G = (V_b == AND). The P and G logic would require 6k switches, 4k memristors, 5k resistors, and 4 drivers for the k-bit block.

Then, when the G(P) value is used for resolving a carry signal or a group signal, a V_{cond} signal must be applied to the G(P) memristor and a V_{set} is applied to the carry-out or GG(GP) memristor. For example, GP=P0P1P2P3. Thus, to resolve the GP memristor, a V_{cond} signal would be applied to each of the P memristors and the p terminals would be sensed. The GP memristor would be applied a V_{set} signal that is gated by a series of 4 switches, each driven by one of the P memristors. The group generate signal for each 4-bit block is computed as GG = G3 + P3G2 + P3P2G1 + P3P2P1G0. This can be implemented in the MAD context with a parallel set of switches in series as shown in Figure 69.

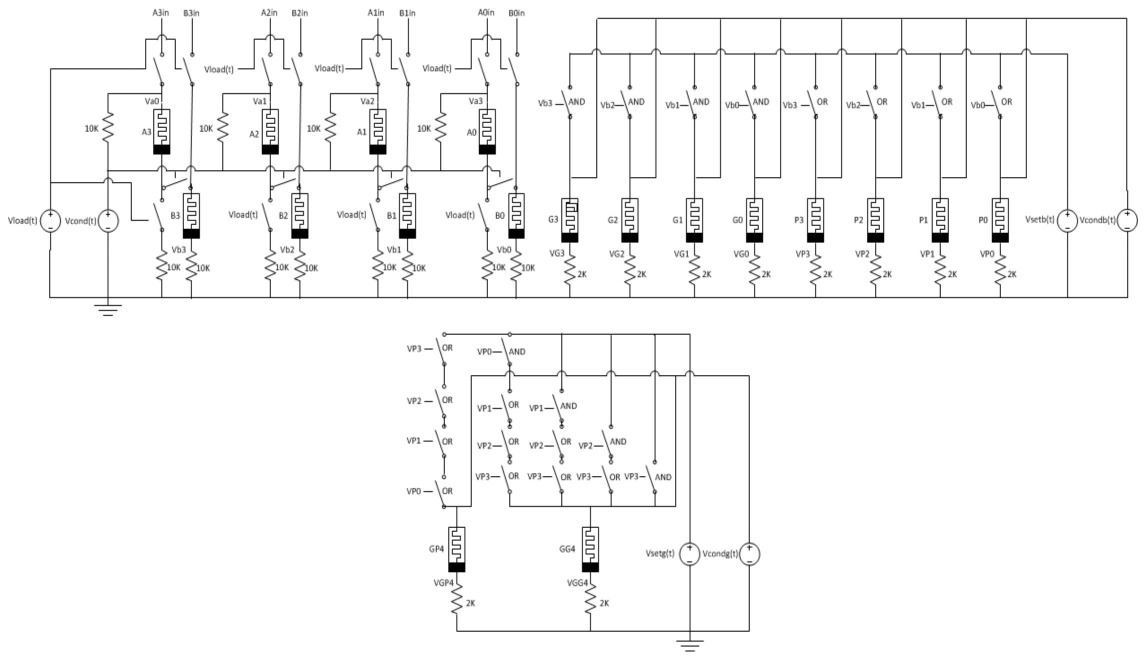


Figure 69: Baseline implementation of MAD GG and GP logic for a 4-bit block in a carry lookahead adder

Each path from V_{set} to the memristor represents one of the AND statements above. By placing them in series, if any of the paths resolves to true, the group generate signal memristor will be set to 1, achieving the OR functionality. The GP and GG logic adds 14 switches, 2 memristors, 2 resistors, and 2 drivers to the circuitry for a total of 6k+14 switches, 4k+2 memristors, 5k+2 resistors, and 6 drivers.

Instead, consider the design with the P and G signals removed. The voltages sensed on the input memristors can be used directly by the carry signals and GG and GP signals. This bypasses the need for the individual bits's P and G signals while maintaining identical functionality. This also removes a step from the overall delay. The resultant circuitry for a 4-bit block is shown in Figure 70.

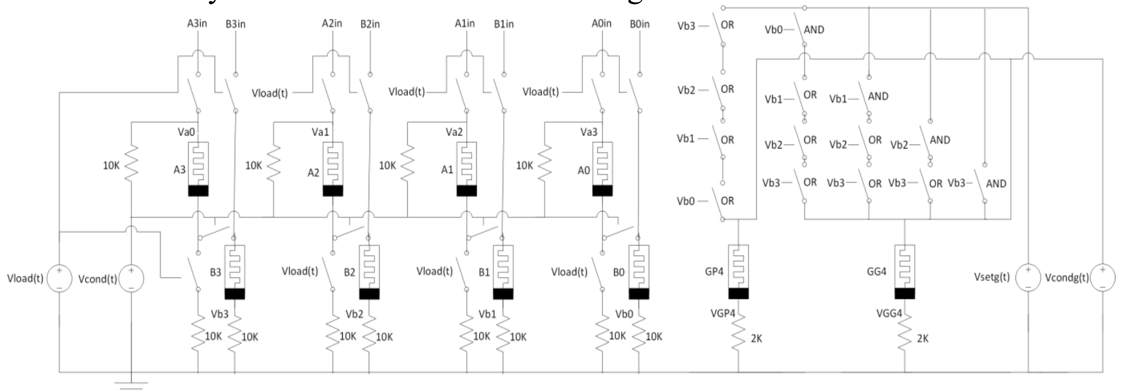


Figure 70: Optimized implementation of MAD GG and GP logic for a 4-bit block in a carry lookahead adder

A V_{cond} signal is still applied to the original memristors A and B. Now, the voltage at the p terminal of the B memristor is used as the driver for the gate on the V_{set} signal to the output carry and group signals directly. Recall that each P_i bit is equivalent to A OR B and is being bypassed in the MAD context. Thus, in the first step of execution, V_{cond} is still applied to all of the input memristors A_i and B_i in each full adder. The p-terminals of each B memristor are then sensed to represent P_i and drive the V_{set} signal on the switches for the group propagate memristor such that GP=(V_{b0} == OR) (V_{b1}== OR) (V_{b2} == OR) (V_{b3} == OR).

A similar translation is performed for the group generate signal, relying on the V_{ai} and V_{bi} signals directly rather than intermediate bits' P and G results. These optimizations reduce the total complexity of the GG and GP logic in the 4-bit block to 4k+14 switches, 2k+2 memristors, 3k+2 resistors, and 4 drivers.

The internal carry bits inside the 4-bit block can be resolved in parallel to the GG and GP signals by probing the same terminal on the input B memristors. These can be constructed with the MAD methodology in the same manner as described for the GG and GP signals.

$CO_0 = G0+P0C_{in}$

$CO_1 = G1+P1G0+P1P0C_{in}$

$CO_2 = G2+P2G1+P2P1G0+P2P1P0C_{in}$

This can be rewritten as

$CO_0 = (V_{b0} == AND) + (V_{b0} == OR)C_{in}$

$CO_1 = (V_{b1} == AND) + (V_{b1} == OR) (V_{b0} = AND) + (V_{b1} == OR) (V_{b0} == OR)C_{in}$

$CO_2 = (V_{b2} == AND) + (V_{b2} == OR) (V_{b1} == AND) + (V_{b2} == OR) (V_{b1} == OR) (V_{b0} == AND) + (V_{b2} == OR) (V_{b1} == OR) (V_{b0} == OR)C_{in}$

Figure 71 shows the resulting circuitry for these carry signals in a 4-bit block.

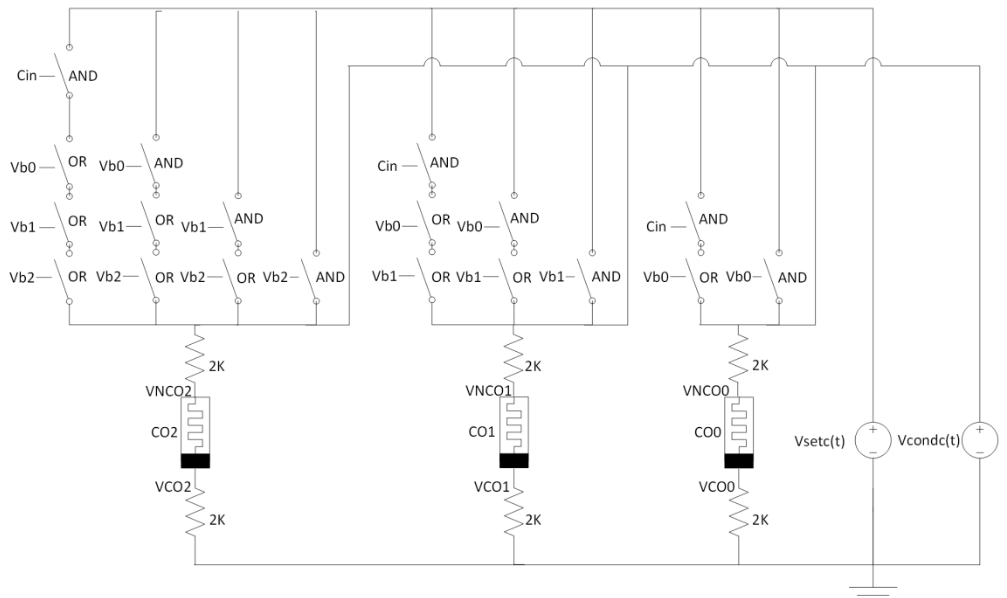


Figure 71: Optimized implementation of MAD carry logic for a 4-bit block in a carry lookahead adder

The sum logic presented for the MAD full adder is reused in this design and can be seen in Figure 72.

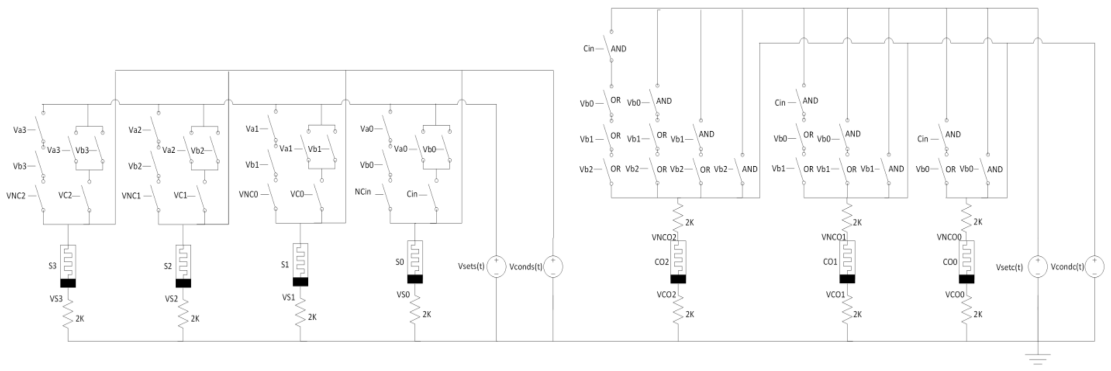


Figure 72: Optimized implementation of MAD sum logic for a 4-bit block in a carry lookahead adder

Note that the same optimization performed to remove the need for G and P memristors, cannot be done for carry-signals. The carry signals necessarily need to be stored into an intermediate memristor. This is because both the carry signals and their inverse are required for resolving the sums. In order to obtain the inverse carry-in signal, the n terminal of the carry-in signals will also be sensed when V_{cond} is applied to the carry memristor. Thus, these memristors are required. For the GG and GP signals, the inverse of the G and P signals were not required so their memristors could be removed.

This process can be extended to implement a 16-bit adder. All of the 4-bit block output G and P signals are resolved in parallel in the first step of execution as described. Then, in the second step, the 16-bit GG and GP signals, the sum signals in the first 4-bit block, and the input carry signals for the other 4-bit blocks ($C4_{in}$, $C8_{in}$, and $C12_{in}$), can be resolved. The carry bits and group bits are resolved using the same circuitry as shown for the initial block but with the switch drivers modified to represent the respective inputs to the equation. In the third step, $C4_{in}$, $C8_{in}$, and $C12_{in}$ are passed into their respective blocks and the carry signals internal to these blocks are resolved. In the fourth and final step, all remaining sum signals for the 16-bit addition are resolved. The complete schematic for the 16-bit adder is shown in Figure 73.

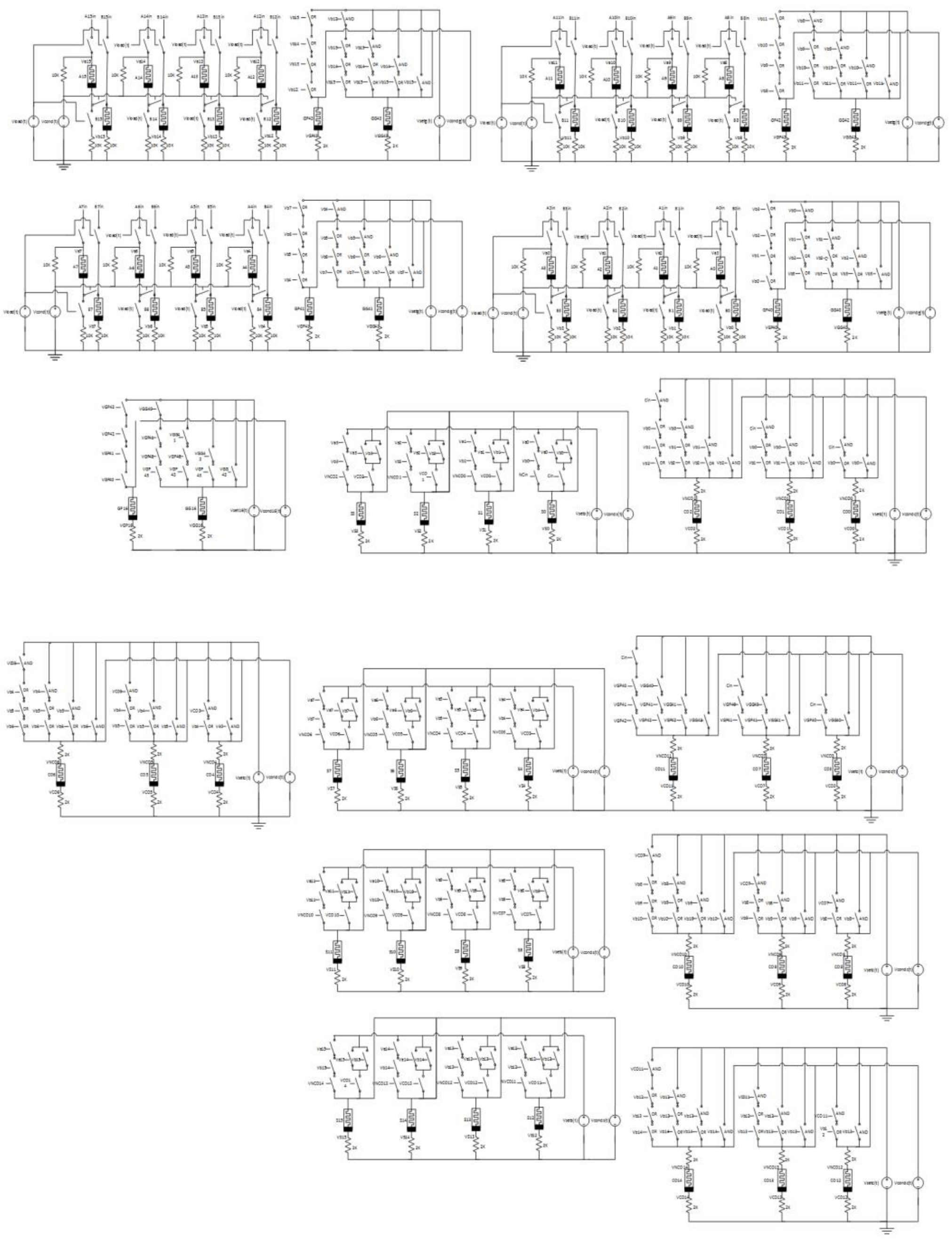


Figure 73: Optimized 16-bit MAD carry lookahead adder

The total delay for a 16-bit carry lookahead addition is 4 steps. A waveform showing the adder's behavior for the addition 0x0 + 0xFFFF with C_{in} = 1 across 4 steps is shown in Figure 74.

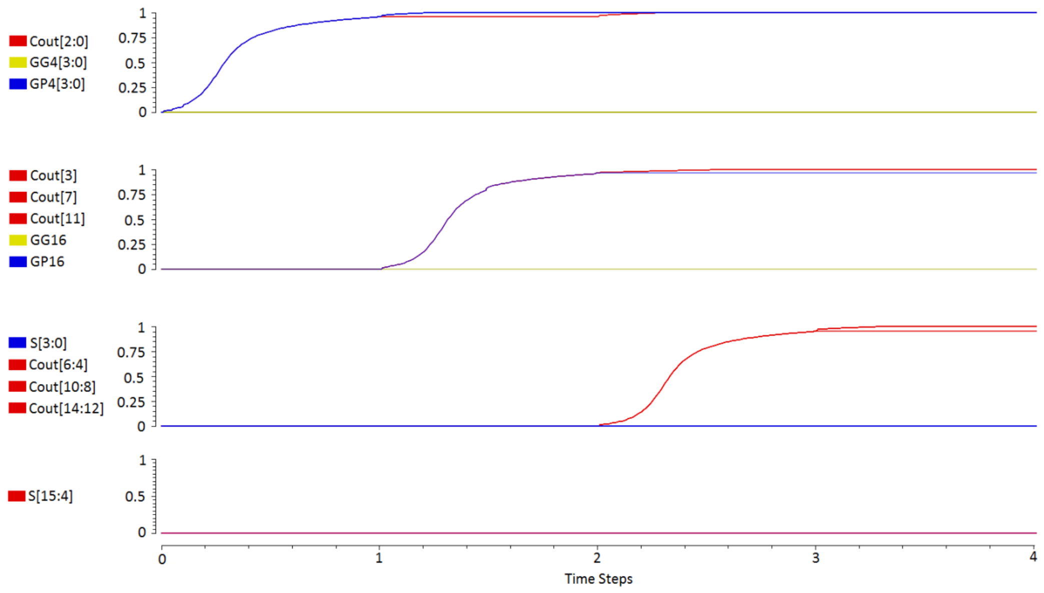


Figure 74: 16-bit MAD carry lookahead adder execution for 0x0 + 0xFFFF

In the first step, the GG and GP bits for each block, and the internal carry signals in the first block, are resolved. The GG values all correctly resolve to 0 since all of the bits' ANDs fail, while all of the GP values resolve to 1 since all of the bits' ORs pass. In the second step, $C4_{in}$, $C8_{in}$, $C12_{in}$, and the 16-bit group propagate GP16 all resolve to 1 and GG16 resolves to 0 by sensing the GG4 and GP4 signals resolved in step 1. In step 3, all remaining carry signals resolve and in step 4, all remaining sum signals resolve.

In general, for an N-bit addition with block size=log(N) blocks, the MAD implementation takes log(N) steps. This is much faster than previous approaches. Also, the MAD design can be pipelined. Once the carry and sum signals have been resolved and propagated from the first 4-bit block, a new addition can begin. Thus, after the first two steps of execution, the block can be reset and loaded with the next inputs. The reset and load take two steps total so a block can begin a new addition every four steps.

Carry Lookahead Adder Analysis and Comparison

The complexity and delay comparisons for the discussed carry lookahead implementations and the CMOS design are given in Table 22.

Table 22: 4-bit and 16-bit delay and complexity comparisons for carry lookahead adders

4-bit Block	**Delay -Cout**	**Delay-Sum**	**Complexity**
Conventional CMOS	6	7	300 MOSFETs
IMPLY	16	20	77 memristors 90 switches 77 drivers
Hybrid-CMOS	7	8	94 memristors+ 40 MOSFETs
Threshold Gate	1	3	40 memristors 8 GOTO pairs 8 MOSFETs
MAD	1	2	17 memristors 73 switches 6 drivers

16-bit Adder	**Delay -Cout**	**Delay-Sum**	**Complexity**
Conventional CMOS	10	11	1212 MOSFETs
IMPLY	22	26	348 memristors 370 switches 348 drivers
Hybrid-CMOS	14	21	426 memristors+ 176 MOSFETs
Threshold Gate	3	5	192 memristors 56 GOTO pairs 32 MOSFETs
MAD	2	4	73 memristors 325 switches 10 drivers

The proposed 16-bit IMPLY implementation has the highest delay out of all of the implementations. However, it offers significant improvement in terms of delay as compared to prior work [16], reducing the delay for the final carry-out and sum bits of the 4-bit block from 25 and 32 to 16 and 20. This improvement magnifies as the size of the adder increases. In fact, the presented optimized 16-bit adder has the same delay as the 4-bit adder in prior work for both the carry-out and sum. The two designs cannot be compared in terms of area because this was not noted in the prior work. In comparison to the traditional CMOS implementation, the IMPLY implementation reduces the complexity from 1212 MOSFETs to 348 memristors and the accompanying switch and driver circuitry. So, although the delay increases by about 2.5x, the number of components is reduced significantly. As the number of consecutive additions increases, the delay is amortized due to the pipelining capabilities of the design.

The proposed hybrid-CMOS approach offers identical delay to traditional CMOS except for delay due to signal buffering. The CMOS usage vastly improves over the traditional CMOS implementation from 1212 MOSFETs to just 176 MOSFETs plus 426 memristors. This is less than half the number of components and the memristor components are much smaller than the MOSFETs they replace. This produces a two-fold improvement in area for the hybrid-CMOS design. In comparison to the IMPLY approach, the hybrid-CMOS approach is faster, but the differential decreases as the adder width increases because of increased signal buffering. Although the complexities cannot be compared directly, the hybrid-CMOS design requires about 50% of the number of components of the IMPLY design.

The delay and the complexity of the threshold gate design are lower than the IMPLY, hybrid-CMOS, and CMOS designs. The use of MOSFETs is very low and the number of memristors is also reduced by combining terms with varying thresholds into the same gate. The delay only increases by 2 when the adder quadruples in size, producing the carry-out bit in just 3 steps for the 16-bit adder.

The delay of the MAD design is the lowest compared to all other evaluated designs. This is partially due to the inherent characteristics of the MAD approach, and partially due to optimizations. For example, by transforming the traditional Boolean logic in the CMOS design and leveraging MAD methodologies, the P and G calculations were eliminated. The delay is less than 1/5 the delay of all of the other designs, beating out the IMPLY design by 10x and 6x for the carry-out and sum signals respectively. The MAD design also requires significantly less area. In comparison to the IMPLY approach, the MAD design reduces the number of drivers from 504 to 10. This is because all of the components, which are set or sensed in the same step can share the corresponding driver in the MAD implementation. The IMPLY approach uses drivers extensively, one per memristor, plus additional drivers for connections across components whereas the MAD approach does not. The number of memristors in the MAD design is reduced to almost 20% of IMPLY while maintaining about the same number of resistors and switches. In comparison to the hybrid-CMOS design, the MAD design uses fewer components. The MAD design also does not require any CMOS or buffering, a key downside of the hybrid-CMOS gates. The MAD design reduces the number of components used in the CMOS design to less than 50% and reduces the individual componential area by replacing MOSFETs with memristors.

IMPROVED AND OPTIMIZED MULTIPLIER DESIGNS[5]

This section presents contributions in the design and implementation of memristor-based multipliers. IMPLY, hybrid-CMOS, threshold gate, and MAD approaches to shift-and-add, Booth, array, and Dadda multipliers are explained and analyzed in terms of complexity and delay. Completed schematics and simulations are also given.

[5] L. Guckert and E. E. Swartzlander Jr., "Memristor-based Logic Design for Dadda Multipliers," *ETCMOS Conference,* May 2017.
L. Guckert and E. E. Swartzlander Jr., "Optimized Memristor-Based Multipliers," *IEEE Transactions on Circuits and Systems I: Regular Papers,* Oct. 2016.
This section is an extension of previously published work for which Lauren Guckert is the principal author and responsible for all content.

Shift and Add Multipliers

Memristor-based shift-and-add multipliers are presented for the IMPLY, hybrid-CMOS, threshold, and MAD contexts.

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. Figure 75 shows the schematic for a 4-bit shift register.

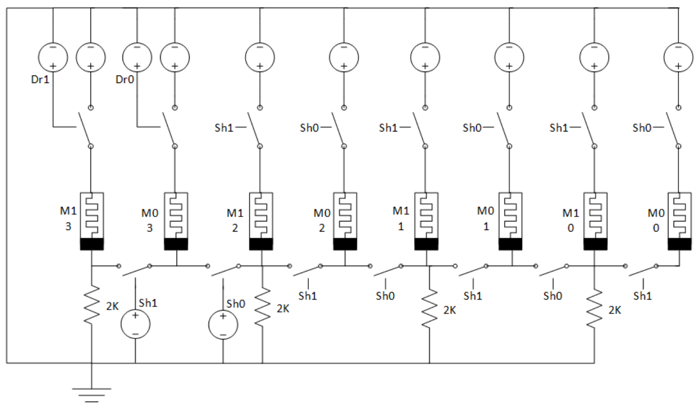


Figure 75: 4-bit IMPLY shift register schematic

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. Note a left shift operation is nearly identical. Recall that a COPY operation is achieved using IMPLY operations by using two cleared 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.

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. This is one optimization performed that was not performed in prior work on shift registers[18]. 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 the shift register produces the registered value's inverse as a "free" byproduct of the 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 is resolved after the first 2 cycles of the shift.

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 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} since all bits act in parallel, 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. The design uses the optimized N-bit ripple carry adder that requires 7N+1 memristors, 7N drivers, 8N-1 switches, and N resistors. It also uses the 2-to-1 N-bit multiplexer that requires 3N+2 memristors, 5 drivers, and 3N+2 switches. An N-bit register requires one memristor and switch per bit, for a total of N each, plus a single driver for loading and sensing the value. 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, 7N+12 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.

This work optimizes the design to reduce this complexity to 7N+1 memristors, 7N drivers, and 8N-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. This multiplier only requires 2 more IMPLY steps per iteration than the ripple carry addition alone.

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. This is a standard optimization that allows the product and multiplier to share the 2N-bit shift register. The multiplicand register has also been eliminated. Since the multiplicand is one of the operands in every addition, 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 can be reduced to require a single memristor per bit and 2 steps. This sequence can be seen in Table 23, with the input B assumed to be 0.

Table 23: Simplified IMPLY shift-and-add multiplexer

Step	**Original Func**	**Original Goal**	**Optimized Func**	**Optimized Goal**
1	Sel imp S0	NOT Sel	A imp NOT Sel	NOT ASel
2	S0 imp S1	Copy Sel	NOT Sel imp out	ASel + B(NOT Sel)
3	A imp S0	A NAND Sel		
4	B imp S1	B NAND NOT sel		
5	S0 imp out	A AND Sel		
6	S1 imp out	ASel + B(NOT Sel)		

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.

The steps for the multiplexer can be incorporated into the first steps of the standard addition. Through Boolean manipulation, 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 24 for the first bit of the multiplication.

Table 24: Steps for one iteration of an IMPLY shift-and-add multiplier for bit 0

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

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. Thus, none of the reset steps contribute to the critical path.

So far, optimizations have been made to the circuit implementation at a low level, but the high-level design itself has not been changed. However, the driver characteristics of IMPLY can be leveraged to optimize the shift-and-add multiplier to take a very different form than traditional CMOS. For example, the shift registers can be removed entirely. 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 IMPLY 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_i 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 Pi[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, sum and product memristors can be removed. 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_i 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 - the result of one iteration of the addition is essentially inserted directly back into the adder as an operand in the subsequent iteration. Due to the driver-based nature of IMPLY logic, this can be achieved directly in the final stage of iteration i. Rather than copying S_i[7:0] from the adder into P_i, 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 S_b is being computed, rather than applying V_{set} to the S memristor for bit b, V_{set} is applied to the B operand in bit b-1. Now, at the end of iteration i, {0,Si[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. The delay is maintained at $2N^2 + 21N$ but the modified steps of execution are shown in Table 25.

Table 25: Optimized steps for the IMPLY shift-and-add multiplier

Step	Functionality	Goal
11		FALSE(M0,M1,M2,B)
…	…	…
22	(C NAND (AXORB)) imp ((C+ (A XOR B)) imp 0) imp 0 =SUM	M3->B(i-1)
23		FALSE all except inputs

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.

Now, 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 Figure 76.

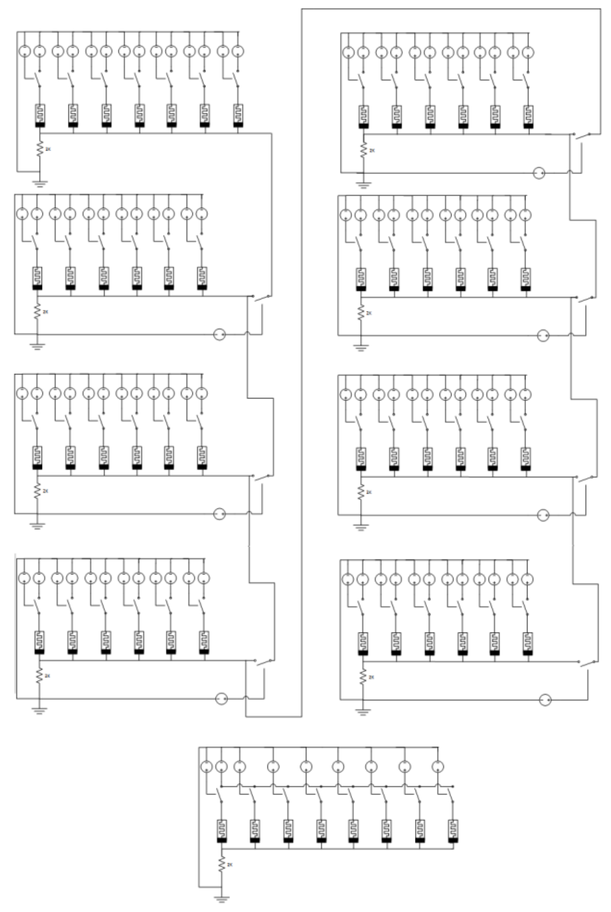


Figure 76: IMPLY shift-and-add multiplier schematic

The top two columns of the schematic are the ripple carry adder and the bottom row implements the multiplier register. Each memristor in the register is logically disjoint so that it can be sensed in its particular iteration. A simulation waveform for the execution of the 8-bit shift-and-add multiplier for the multiplication 0xAA * 3 = 0x1FE is shown in Figure 77.

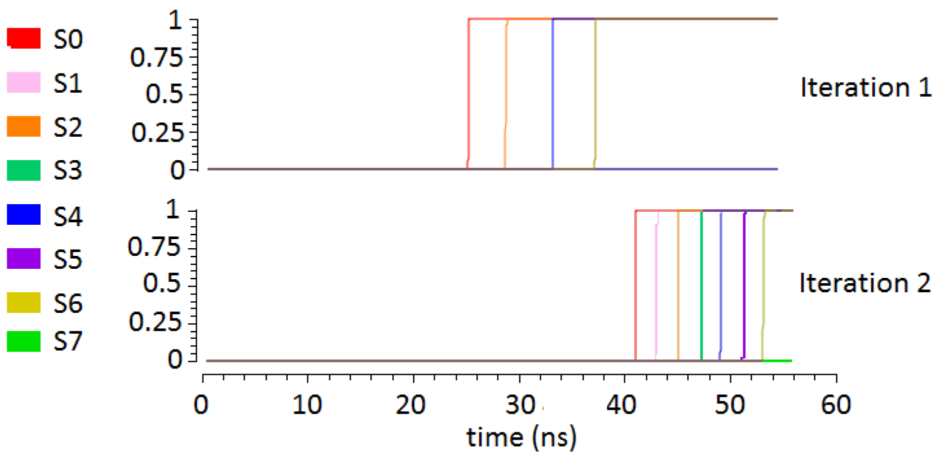


Figure 77: IMPLY shift-and-add multiplier execution for 0xAA * 3 = 0x1FE after two iterations

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 0xAA is added to 0. The result is 0xAA but since the sum is shifted as part of the addition, the sum bits S[7:0] stored in operand B are 0xAA >> 1 = 0x55. 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 B operand 0x55 is added to 0xAA. The sum, 0xFF is shifted and the sum in the B operand becomes 0xFF >> 1 = 0x7F.

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 so 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.

HYBRID-CMOS SHIFT AND ADD MULTIPLIER

A hybrid-CMOS design for a shift-and-add multiplier is presented that requires 32N MOSFETs, 32N memristors, and a delay of $3N^2+13N$. First, a novel shift register is presented which requires 6N gates, or 12N memristors. A 4-bit schematic for a hybrid-CMOS shift register is shown in Figure 78.

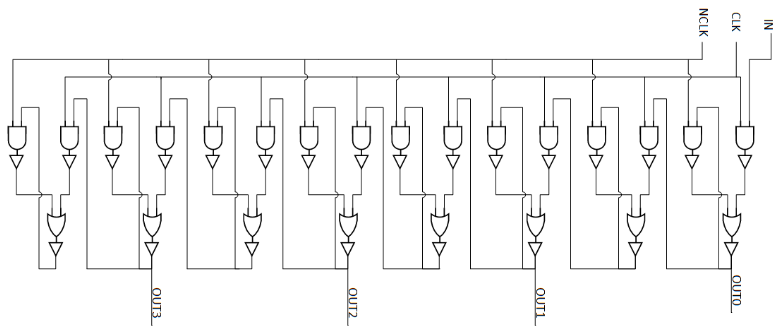


Figure 78: 4-bit hybrid-CMOS shift register schematic

Shift registers exhibit high gate concatenation and instability due to the feedback nature of the individual latches. Thus, each individual AND and OR gate requires a buffer, increasing the complexity to 12N memristors + 24N MOSFETs. Each shift requires 6 steps as shown in Figure 79 for the input 0x9.

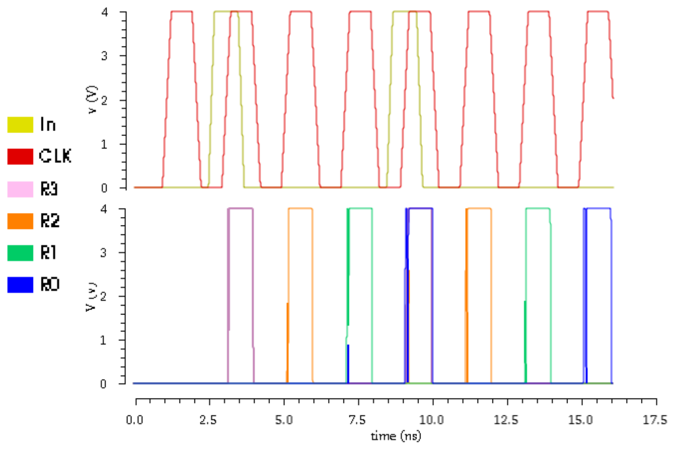


Figure 79: Hybrid-CMOS shift register execution for the input 0x9

This shift register is used in conjunction with the previous section's optimized hybrid-CMOS ripple carry adders and multiplexers to produce the proposed design in Figure 80.

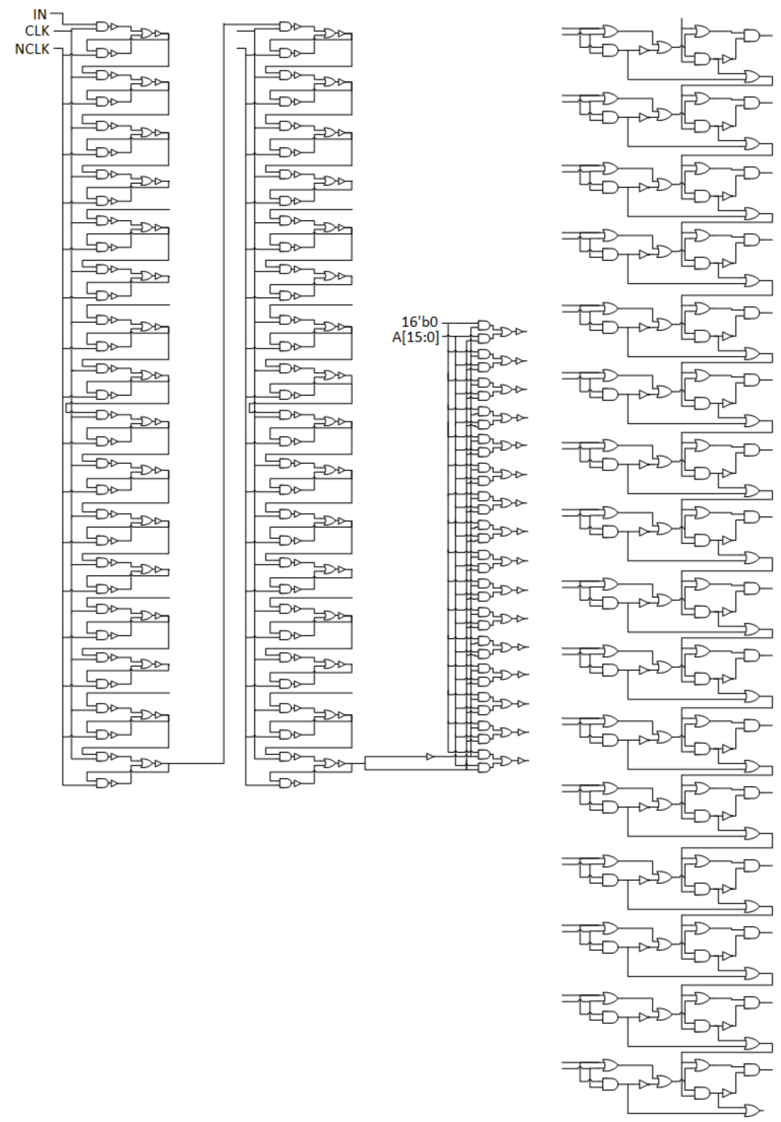


Figure 80: Hybrid-CMOS shift-and-add multiplier schematic

The top two rows of the schematic are the shift registers for the multiplier and product. The bottom of the schematic holds the multiplicand, the 16-bit ripple carry adder, and the multiplexer. The ripple carry adder requires 14N memristors and 12N MOSFETs and the mux requires 6N memristors and 4N+2 MOSFETs (for output buffering and the inverse select line). The total complexity is 32N memristors and 40N+2

MOSFETs. The functional behavior of this multiplier for the first iteration of the operation 0x5555 x 3 = 0xFFFF is given in Figure 81.

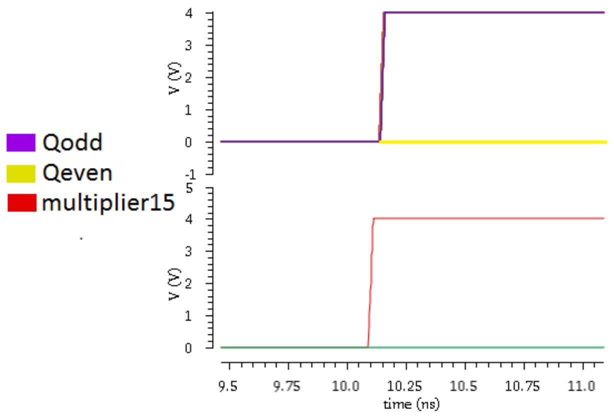


Figure 81: Hybrid-CMOS shift-and-add multiplier execution for 0x5555 * 3 = 0xFFFF after one iteration of the multiply

The result of the first iteration is 0x5555. After the shift at 10.1ns, the top bit of the multiplier shift register holds a 1 and the bits of the product register have shifted.

THRESHOLD GATE SHIFT AND ADD MULTIPLIER

The proposed threshold gate design requires 36N memristors, 3N NOT gates, 11N GOTO pairs and N^2+4N gate delays. The majority of the area comes from the shift register which requires 26N memristors, 2 NOT gates, and 9N GOTO pairs. The shift register consists of a series of 2N flip flops. The most significant N flip-flops require the ability to load a value in addition to shifting a value. The least significant N flip-flops only require the ability to shift a value. The diagram for the implementation of these two flip-flops is given in Figure 82.

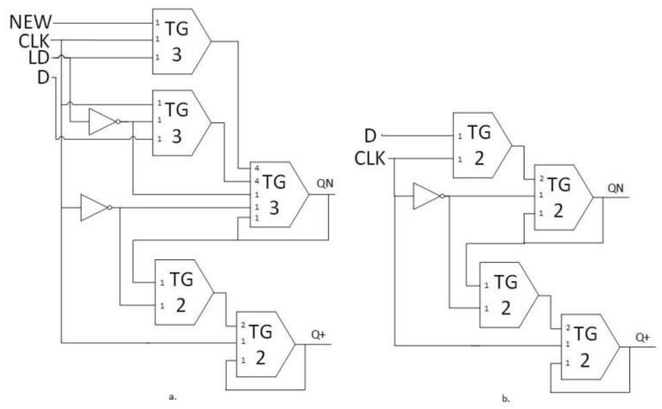


Figure 82: Diagram for a flip-flop designed from threshold gates a) with and b) without load functionality

The flip-flop which does not use the LOAD signal requires a NOT gate, 4 GOTO pairs, and 10 memristors while the flip-flop which incorporates a LOAD signal requires an additional NOT gate, GOTO pair, and 6 memristors. In the context of the shift register, the NOT gates do not need to be repeated in each cell, and thus the entire shift register only requires 4 MOSFETs. The simpler flip-flop has a delay of 2 gates while the more complex one has a gate delay of 3 due to the additional NOT gate.

The shift register is used in conjunction with a threshold-based ripple carry adder and multiplexer to complete the shift-and-add multiplier design. The previous section presented a ripple carry adder which requires N+1 gate delays and 2N gates. It also presented a mux which requires 2 threshold gates per bit and a CMOS NOT gate. This work proposes a design which implements both the ripple carry adder and the multiplexer with a total of N+1 delays and 2N gates. The design uses a 1-bit adder as shown in Figure 83.

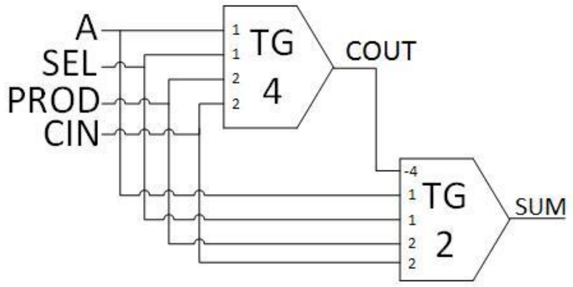


Figure 83: Threshold gate 1-bit adder design optimized for use in a shift-and-add multiplier

This adder implements both the multiplexing of the multiplicand A as well as the addition of this operand with the running product and carry in C. Note that the multiplexer logic outputs a 0 if either A or the select line is a 0. The output is only a 1 when both A and the select line are 1. Thus, the original design for a full adder is simply enhanced to weight the A and select line inputs each with half the weight of the other inputs. In this way, they will only achieve the equivalent weight of the other inputs if both are a '1', essentially performing an 'AND' within the threshold gate itself.

The adder requires only 9 memristors and 2 GOTO pairs and maintains the delay of N+1 delays for an N-bit addition. The total complexity of the threshold-based design is 35N memristors, 4 MOSFETs, and 11N GOTO pairs. The delay of each iteration is N+1 gate delays through the adder and 3 gate delays in the shift register. For an N-bit multiplication, the total delay is N^2+4N gate delays.

MAD GATE SHIFT AND ADD MULTIPLIER

A MAD gate design for a shift-and-add memristor-based multiplier is proposed. First, the design minimizes the delay internal to each full adder in the ripple carry adder. This design uses the optimized full adder which requires 4 memristors, 5 resistors, 13 switches, and 4 drivers and only 1 step as the baseline. This work further improves on this by leveraging information about the shift-and-add multiplier context. The resultant full adder is shown in Figure 84.

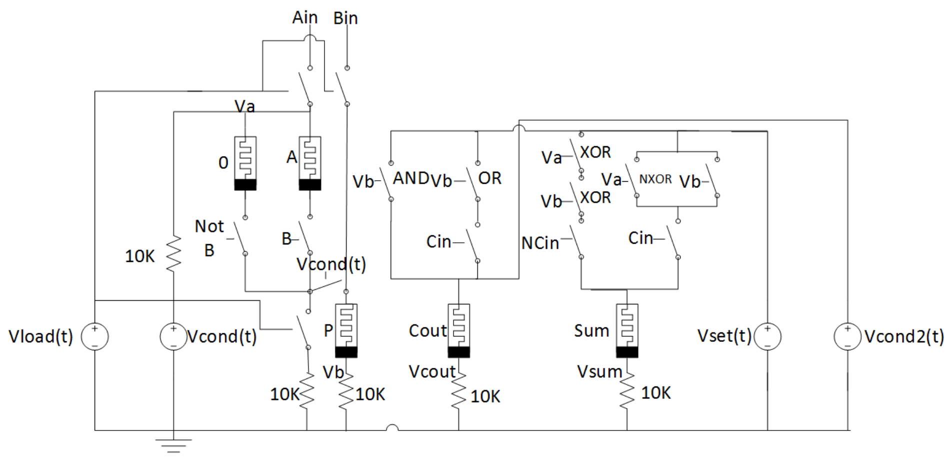


Figure 84: MAD full adder for a shift-and-add multiplier

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. 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.

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 resolve the final carry-out and sum memristors as normal. 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'. This full adder can now be connected in the same way the ripple carry adder was.

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 Figure 84 for the sum memristor, will now be resolving sum_{i-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 achieves the shift functionality as part of the addition steps. This is possible because the sum memristor and its drivers are completely independent from the rest of the circuit.

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 another N memristors by eliminating the sum memristors. The modified MAD full adder is shown in Figure 85.

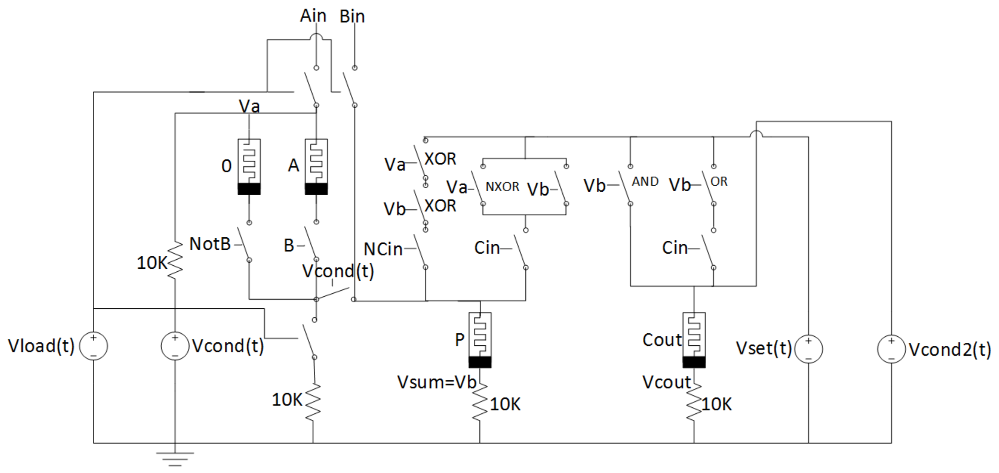


Figure 85: Optimized MAD full adder for a shift-and-add multiplier

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 Figure 86.

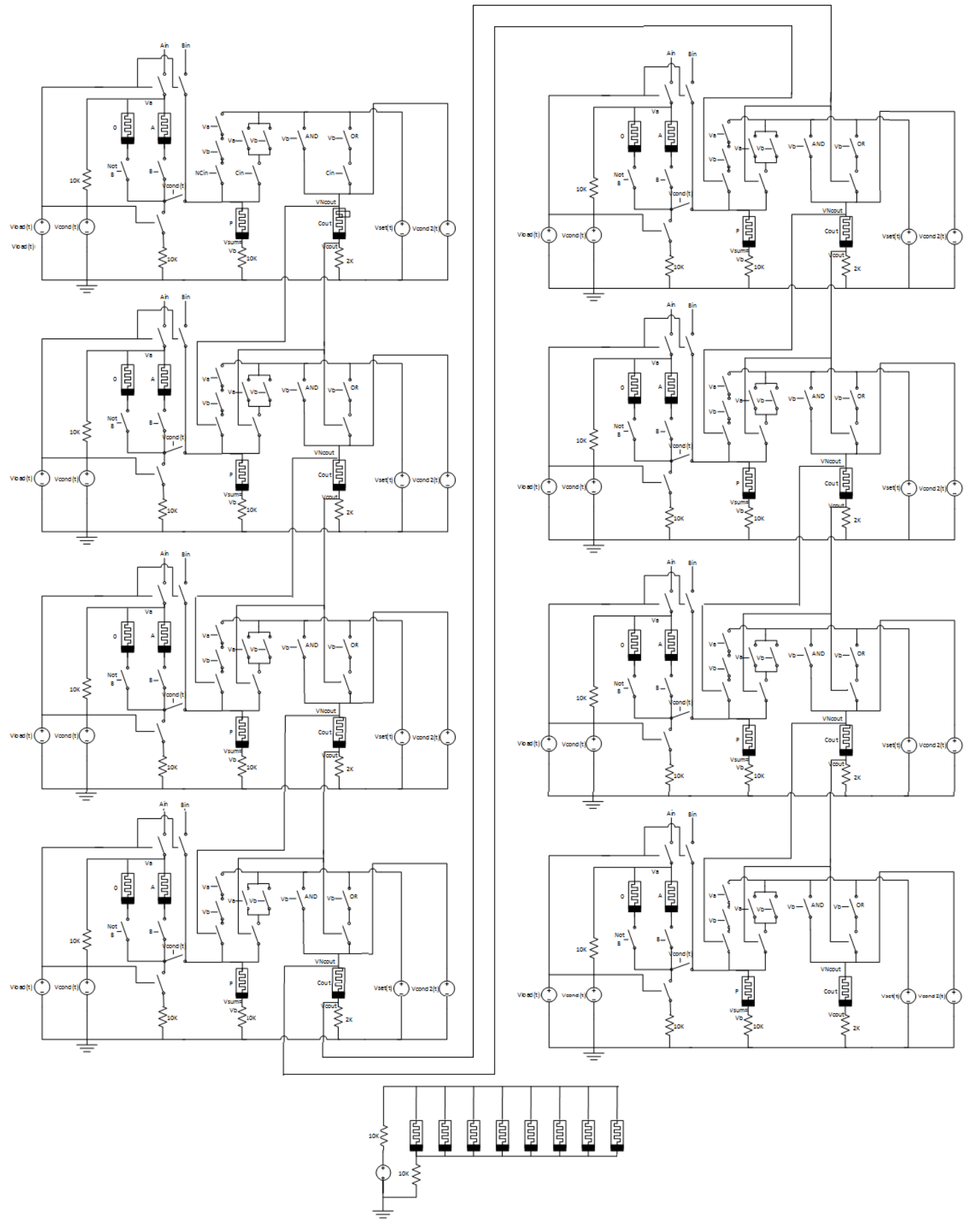


Figure 86: 8-bit MAD shift-and-add multiplier

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 (same as the standard MAD ripple carry adder) 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 throughput of the MAD shift-and-add multiplier is now every 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 N drivers. A waveform showing this ability for 0x56 * 3 is shown in Figure 87.

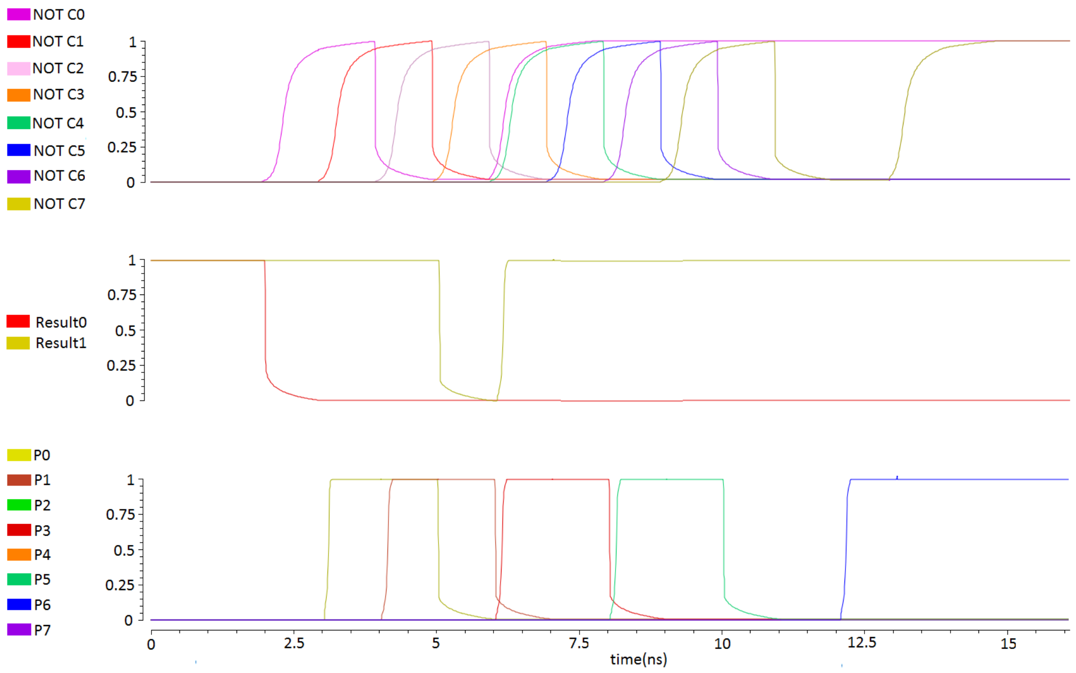


Figure 87: MAD shift-and-add multiplier execution for 0x56 * 3 for two iterations

The waveform shows two full iterations of the multiplication. In the first iteration, 0x56 is added to 0. The sum and inverse carry out memristors are each resolved 1 step after the previous bit from t=2ns to t=9ns. 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 0x2B rather than 0x56. 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=2ns. At t=3ns, the sum memristor in bit 0 is set. At t=4ns, the carry-out memristor is cleared. At t=5ns, the second iteration begins, adding P=0x2B to the multiplicand A. This results in all of the inverse-carry-outs going high and the P memristors to 0x81. Given the shifted sum result, the product register has

0x40 at the end of the second iteration and bit 1 of the storage product register has a 1. Two complete iterations of the multiplication complete at t=13ns.

SHIFT AND ADD MULTIPLIER ANALYSIS AND COMPARISON

The proposed designs significantly decrease the delay and complexity of the multipliers as compared to the traditional CMOS designs. A full comparison of the delay and complexity of each design are given in Table 26.

Table 26: Delay and area comparisons of the proposed shift-and-add multipliers

	Traditional CMOS	Proposed IMPLY	Proposed Hybrid-CMOS	Threshold Gates	Proposed MAD
Delay	$2N^2+23N$	$2N^2+21N$	$3N^2+13N$	N^2+4N	N^2+N
Pipelined Delay	N/A	24N	N/A	N/A	4N
Complexity	132N+6 MOSFETs	7N+1 memristors 7N drivers 8N-1 switches	32N memristors 40N+2 MOSFETs	35N memristors 4 MOSFETs 11N GOTO pairs	5N memristors 3N+2 drivers 14N switches

The hybrid-CMOS design is the only design that has a greater delay than the CMOS design for large N. This is explained by the buffering necessary in the shift register and the ripple carry adder carry propagation. The proposed IMPLY design follows closely with traditional CMOS in terms of delay but each improves by a few cycles each iteration. The IMPLY operation is only 2N fewer steps, a competitive delay given the serialized nature of IMPLY steps. The threshold gate reduces the delay by over 2X from $2N^2+23N$ to N^2+4N. The proposed MAD design further reduces the delay to only N^2+N steps, a significant savings, especially as N increases. This is over 2x fewer steps than the optimized IMPLY, hybrid-CMOS and CMOS designs. When the designs are fully utilized, both the IMPLY and MAD benefit from the ability to pipeline consecutive additions. For both proposed designs, the latency reduces from O(N^2) to O(N). For the IMPLY implementation, the throughput of an N-bit multiplication reduces from one multiplication every $2N^2$+21N steps to 24N. For the MAD implementation, the throughput is 4N steps per multiplication - only 64 steps for an entire 16-bit multiplication. This is a hugely significant improvement over traditional CMOS designs which requires 880 steps. The hybrid-CMOS, threshold, and traditional CMOS design cannot perform pipelining, thus the latency remains the same.

In terms of area, the proposed MAD designs requires fewer components than the other designs, even the IMPLY approach which is known to be lowest area in general. The number of memristors and drivers in the MAD design reduces by 43% as compared to the optimized IMPLY design. This comes at the cost of more switches as compared to IMPLY, but the number of switches is still significantly fewer than the baseline. Both of the designs significantly reduce the number of components to about 1/6 of the CMOS design. The proposed hybrid-CMOS and threshold gate approaches require more complexity but are similar to each other and still reduce the componential area to about 50% of CMOS.

IMPLY AND MAD BOOTH MULTIPLIERS

Both the IMPLY and MAD designs are 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.

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(NOT M_i) $M_{(i-1)}$ + A_i'M_i NOT $M_{(i-1)}$, where Ai' represents the 2's complement of A for bit i and M_i and $M_{(i-1)}$ represent the select lines. The multiplexer executes the steps in Table 27.

Table 27: Modified steps for a 2-bit IMPLY multiplexer for Booth multipliers

Step	Functionality	Goal
1	$M_{(i-1)}$ imp T0	NOT $M_{(i-1)}$
2	T0 imp T1	Copy $M_{(i-1)}$
3	M_i imp T1	NAND (M_i (NOT $M_{(i-1)}$))
4	M_i imp T2	NOT(M_i)
5	T2 imp T0	NAND((NOT M_i) $M_{(i-1)}$)
6	A' imp T1	NAND A'M_i (NOT $M_{(i-1)}$)
7	A imp T0	NAND A(NOT M_i) $M_{(i-1)}$
8	T0 imp T3	A(NOT M_i) $M_{(i-1)}$
9	T3 imp T1	A(NOT M_i)$M_{(i-1)}$ + A'M_i (NOT $M_{(i-1)}$)

This multiplexer has been simplified and optimized by rearranging the IMPLY operations to minimize the number 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 improve the throughput of each iteration. Note that steps 1-5 in Table 27 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 24 and the multiplexer result exists in M5 as before. Thus, Booth multiplier only takes a single step more per iteration than the shift-and-add multiplier. Two additional memristors, T0 and T1, 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, M_i and $M_{(i-1)}$ 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.

To accommodate Booth 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, gated by M_i and NOT M_i. This must be modified to accommodate checking the value of two multiplier bits, M_i and $M_{(i-1)}$. Figure 88 shows the modified adder for the Booth multiplier.

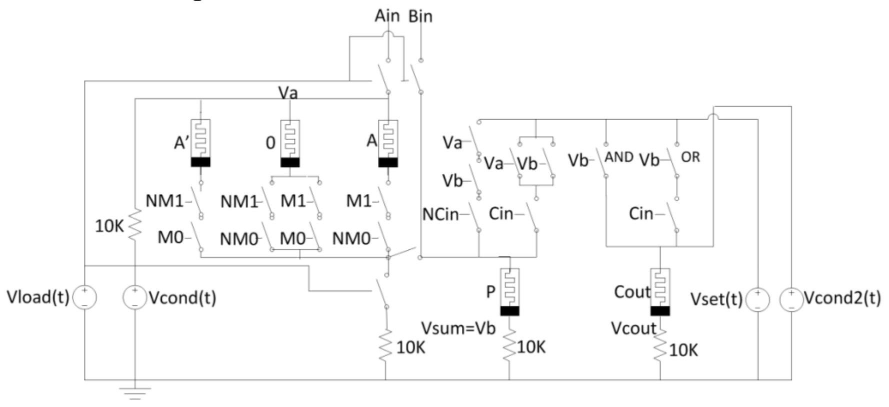


Figure 88: MAD full adder for Booth multiplier

A comparison of the proposed Booth multipliers against the traditional CMOS design is given in Table 28. The same limitations on the comparison still exist as discussed for the shift-and-add multipliers.

Table 28: Delay and complexity comparisons of the proposed Booth multipliers

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

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 T0 and T1. 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. 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 half of the number of CMOS or IMPLY steps.

It can also be pipelined by the same logic as the shift-and-add multiplier to only require a throughput 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.

Array Multipliers

Memristor-based array multipliers are presented for the IMPLY, hybrid-CMOS, threshold, and MAD contexts.

IMPLY ARRAY MULTIPLIER

A memristor-based array multiplier using the IMPLY operation is designed that achieves a delay of 24N-35 steps with a component count of $7N^2$-8N+9 memristors. The array multiplier presented in this work uses the presented optimized ripple carry adder in conjunction with AND gates to create the array structure. Recall the optimized N-bit ripple carry adder has a delay of 2N+19 steps and 7N+1 memristors. An N-bit array multiplier requires N-1 of these N-bit ripple carry adders. Additionally, the design requires N^2 AND gates for resolving the input bit combinations. Lastly, since the inputs to the individual full adders are used more than once, they must be copied from the output of the AND gates before use, requiring 2 memristors per full adder input. Thus, the baseline array multiplier design requires 9(N-1)N + $5N^2$ memristors, or $14N^2$-9N memristors.

The critical path delay traverses through an AND gate and then through both dimensions of the array. Within a row, each full adder propagates its carry bit to the subsequent adder in 2 cycles. However, across rows, each full adder requires the entire full adder delay of 21 steps to produce its sum bit to pass as an input to the subsequent row. Thus, the total delay for the baseline N-bit multiplier is 3 steps for the initial add + 2(N-1)+19 for the first row ripple carry add + 21(N-3) to traverse vertically through the rows + 2N+19 for the final row ripple carry add. The total delay is 25N-24 IMPLY steps.

Prior work developed a 4-bit array multiplier with 3 fewer steps than this baseline but the area was not analyzed [14]

This work performs a series of optimizations to reduce the component count to $7N^2$-8N+9 memristors and delay to 24N-35, a 16% delay improvement over prior work [14]. First, the full adders with only two inputs are converted to half adders. No prior work has proposed a design of a half adder with IMPLY. This work optimizes a half adder to require 8 IMPLY steps and 6 memristors.

Second, the design leverages the fact that each AND gate result is used by exactly one adder. Thus, the memristors and steps required for performing the AND operation can be incorporated into the adders and no copies need to be performed. The steps for a half adder with the AND operation incorporated are given in Table 29.

Table 29: Execution steps for an IMPLY half adder

Step	Functionality	Description
1	B imp 0 -> S0	NOT B
2	A imp (B imp 0)	A NAND B = NP
3	(A imp (B imp 0)) imp 0	A AND B = P *The results of the AND and NAND operations are now both available in P, NP, Q, and NQ
4	P imp NQ -> NQ	P NAND Q
5	P imp NQ imp 0 -> Cout,S1	P AND Q = Cout
6	NP imp Q -> Q	
7	Q imp S1 -> S1	P XNOR Q
8	S1 imp 0 -> Sum	P XOR Q = Sum

A and B represent the inputs to the AND gate. P and Q represent the inputs to the adder. The adder is able to reuse memristors from the AND operation as well as perform initial addition steps before the AND result is resolved. The full adders in the design also inherit this optimization to incorporate the AND operation. By strategically timing the driver circuitry to accommodate these changes, the full and half adders in the proposed design are able to maintain the same area of 7 memristors and 6 memristors after incorporating the AND operation into the functionality. All full and half adders are modified in this way except for the first half adder and first full adder. These adders do not use this technique because their operations are along the critical path and thus the delay is optimized at the cost of increased memristors.

Third, optimizations are performed across the adders which share inputs and outputs to remove the need to copy the outputs. Rather than storing the result of the sum into a resident memristor, each adder directly stores its sum result into a memristor in the next adders that need it. Also, IMPLY operations are rearranged to ensure that operands with a fanout greater than 1 are never overwritten until they are finished being used and can be reset. Lastly, a few small optimizations can be performed on various adders to remove memristors. For example, since bit 0 of the sum is not on the critical path, more memristor reuse can be done. Rather than using 4 memristors and 3 IMPLY steps, the design uses 3 memristors and 4 steps. Similar logic is used to minimize area elsewhere off the critical path. The schematic for a 4x4 array multiplier is shown in Figure 89.

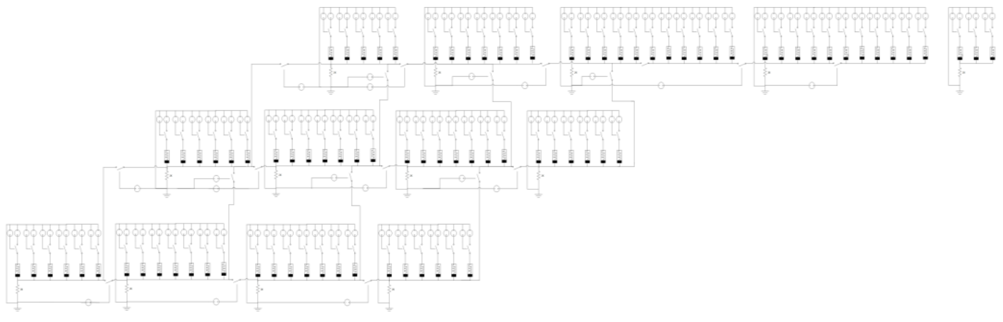


Figure 89: 4-bit IMPLY array multiplier schematic

Each row corresponds to a ripple carry adder with the integrated AND gates except for those on the first row. Each adder is driven by inputs from the adder above it in the array and to the right of it in the array. This is denoted by the driver and switch placement between each bit.

The final design requires $7N^2$-8N+9 memristors for an N-bit multiplier. This is almost half of the baseline requirement, $14N^2$-9N memristors. In terms of delay, the proposed design reduces the delay from 25N-24 to 24N-35. A waveform showing the resolution of each sum bit for the multiplication 0xF * 0x7 is shown in Figure 90.

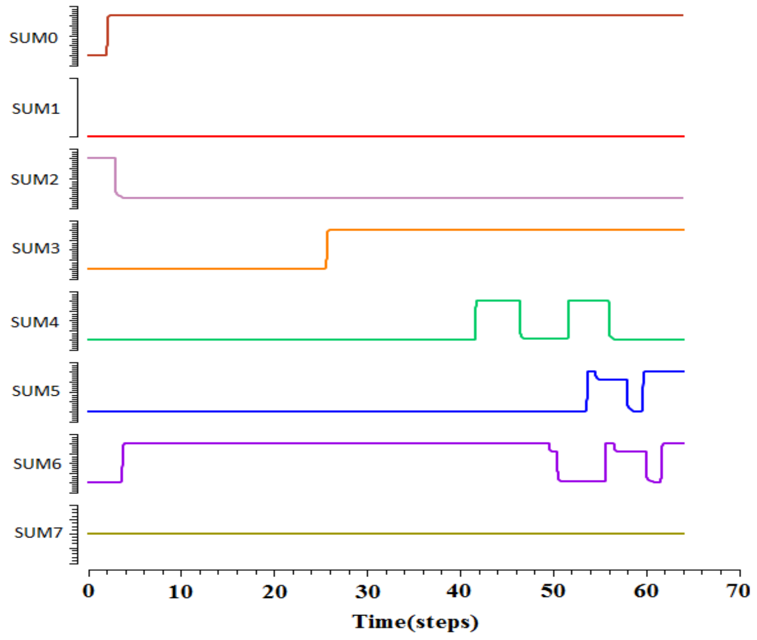


Figure 90: 4-bit IMPLY array multiplier execution for 0xF * 0x7 =0x69

This multiplier can also be pipelined across full adders as described to complete a multiplication every 21 IMPLY steps. Alternatively, by the same reasoning, an N-bit IMPLY-based multiplier can be reduced to just the first two rows of the array if area is of concern. As the bits in the first row complete, they resolve their results into the adders in the second row. As the inputs arrive, the second row begins addition, resolving its results back into the adders in the first row. Execution continues in this toggle manner, requiring the same delay as the original design but reducing the area to only 2 rows of the design, or 14N+8 memristors.

HYBRID-CMOS ARRAY MULTIPLIER

The proposed hybrid-CMOS approach requires a total of $6N^2$-11N+2 MOSFETs + $14N^2$-16N memristors and achieves a delay of 13N-12. The proposed design uses the same array structure as traditional CMOS. A schematic for the hybrid-CMOS based 4x4 array multiplier is shown in Figure 91.

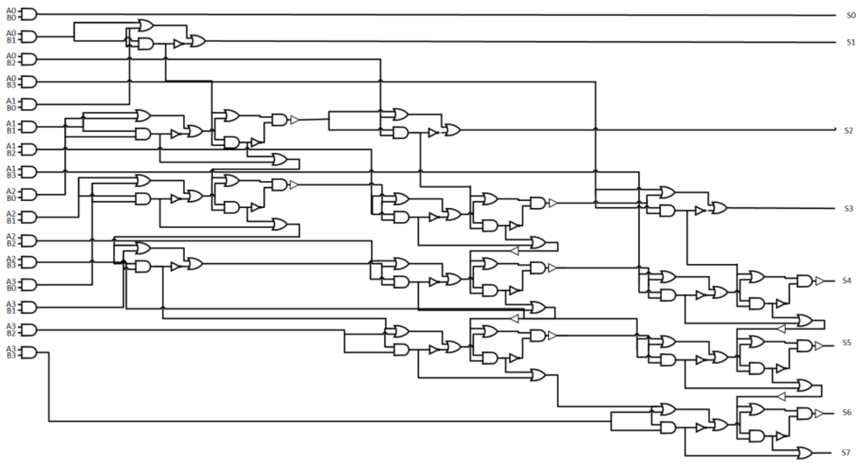


Figure 91: 4-bit Hybrid-CMOS array multiplier schematic

The array multiplier lends itself to the hybrid-CMOS approach because it does not use any NOT gates outside of the adders themselves so minimal CMOS is required. Also, the AND operation can be performed in a single step, producing the inputs to the first adders after less delay than the IMPLY design. Lastly, in this design, the fanout of each AND gate is 1 so no signal buffering is necessary on these gates.

Note that the optimized XOR gate discussed in previous sections is essentially a half adder because the AND operation is performed as part of the XOR operation. Thus, each half adder in this design is implemented using this gate and requires 6 memristors and 1 NOT gate rather than 8 memristors and 2 NOT gates. Through simulation, signal restoration was found to be necessary on all intermediate adder outputs due to gate concatenation. Thus, for an N-bit array multiplier, $2N^2$-5N+2 CMOS buffers are required. The AND gates require $2N^2$ memristors and the ripple carry adders require $14N^2 - 16N$ memristors and $4N^2 - 6N$ MOSFETs. Thus, the total complexity is $6N^2$-11N+2 MOSFETs + $14N^2$-16N memristors.

One drawback of the design is that the signal restoration lies on the critical path. Each propagated carry signal and sum signal must be buffered. Thus, the delay of the ripple carry adders becomes 3N+4 rather than the traditional 2N+4, and the delay through a full adder is 7 rather than 6. Thus, the total delay for an N-bit multiplication is 3N+4 steps through the first ripple carry adder, 7(N-3) steps to traverse the array vertically, and another 3N+4 steps through the final ripple carry adder, for a total of 13N-12 steps. A waveforms showing the resolution of the individual product bits for 0xF * 0x7 is shown in Figure 92.

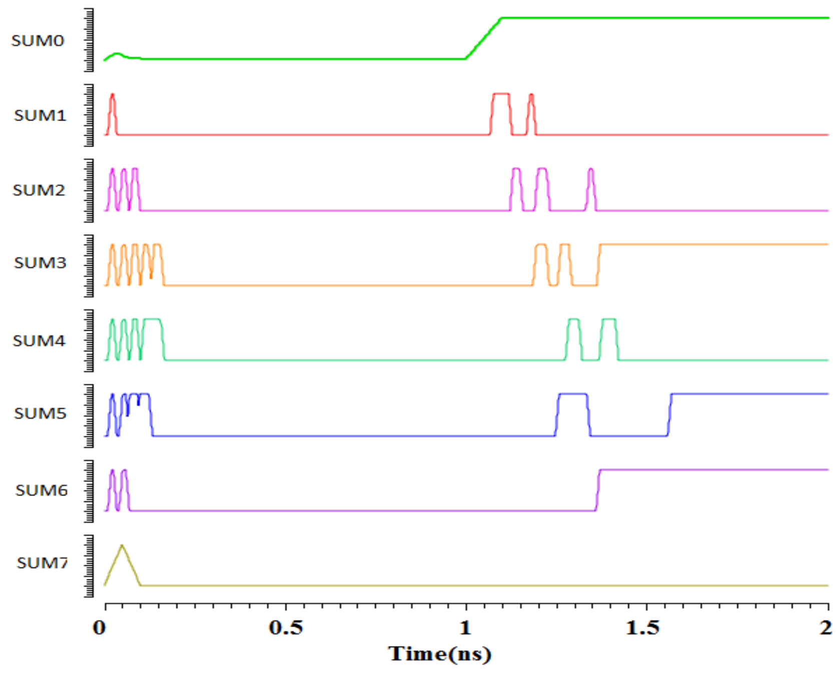


Figure 92: 4-bit Hybrid-CMOS array multiplier execution for 0xF * 0x7 =0x69

Threshold Gate Array Multiplier

The proposed threshold gate design for an array multiplier requires $9N^2$-9N memristors and $2N^2$-4N GOTO pairs with a delay of 4N-3 gate delays. The threshold N-bit ripple carry adder requires N+1 gate delays and 7N memristors and 2N GOTO pairs. Additionally, a threshold gate consisting of 2 memristors and 1 GOTO pair can be used to implement an AND gate. Thus, the baseline complexity for the optimized array multiplier is $9N^2$ -7N memristors and $3N^2$-2N GOTO pairs and the baseline delay is 1+(N+1)+2(N-3)+(N+1) = 4N-3. This work optimizes the implementation of the individual adders to further reduce the area and delay.

First, the half adders for the least significant bit in each row of the array multiplier can be simplified given the fact that they only have two inputs and enhanced to incorporate the AND functionality too. An implementation of this half adder is shown in Figure 93.

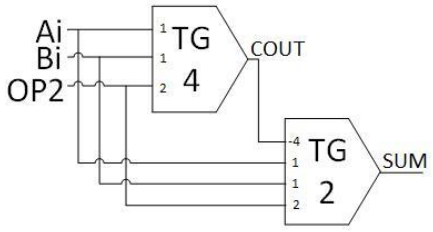


Figure 93: Diagram for a threshold-based half adder in the array multiplier

OP2 represents the output from the previous adder which serves as an input here as usual. A_i and B_i are the two inputs that require an AND to produce the input to the half adder. The AND operation is consumed into the threshold gates in the half adder. The half adder is able to maintain an complexity of 7 memristors and 2 GOTO pairs, removing the need for 2 memristors and a GOTO pair for each AND operation. It also removes a step from the delay. The AND gates can also be integrated into the full adders as shown in Figure 94.

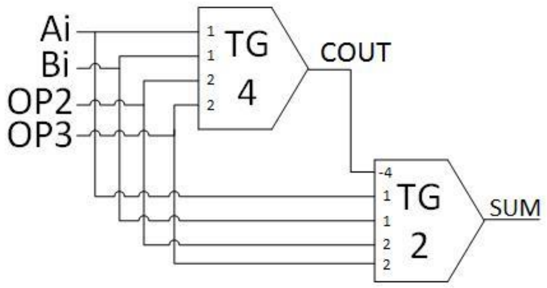


Figure 94: Threshold gate implementation of full adders optimized for an array multiplier

In this case, the total number of memristors does not change but a GOTO pair is removed since the AND is no longer performed externally. For the first N-1 adders which use the result of an AND operation as both their inputs, it is not possible to incorporate both AND operations into a single threshold gate. This is because it is not possible to differentiate which inputs correspond to which AND gates with only weights.

For example, if you wanted to integrate the second AND into the threshold gate, C_i AND D_i, there are no possible weights to achieve this. If they were given the same weights as A_i and $B_{i,}$ the gate would not be able to differentiate between the case where only A_i and B_i were 1 (should results in a 0 and 1 output for the ANDs) and the case where only A_i and C_i were 1 (should result in a 0 output for both ANDs). Both scenarios would behave equivalently to the case where one of the ANDs resolved to 1. Thus, for each of the initial N-1 adders, one of the inputs uses an external threshold gate to perform its AND operation.

The total complexity for the array multiplier is reduced from $9N^2$ -7N memristors and $3N^2$-2N GOTO pairs to $9N^2$ -9N memristors and $2N^2$-4N GOTO pairs. The total delay remains 4N-3 gate delays since the critical path delay cannot remove the need for the AND gate as described above.

MAD GATE ARRAY MULTIPLIER

The proposed MAD array multiplier design requires $5N^2$-2N+2 memristors plus driver circuitry and only 3N-4 steps for the multiplication. The optimized ripple carry adder coupled with N^2 MAD AND gates would require a total of (N-1)(4N memristors + (3N+1) drivers + 13N switches) + N^2(3 memristors + 1 switch) + 3 drivers as a baseline. However, both the delay and area can be reduced in the context of an array multiplier.

First, N of the full adders can be replaced with half adders. The full adder and half adder take the exact same form except the half adder has a simplified switch circuitry for the C_{out} and sum signals. Specifically, the C_{out} memristor only requires 1 switch for the AND and the sum memristor only requires 2 switches for the XOR. Since there are N half adders, this removes 6N switches from the design.

Next, the full adders can be optimized to store their sum into an input memristor in the adder which uses this value as an input as previously described. Thus, each sum will be resolved into either the A or B input in the "next" full adder in the chain. Now, every intermediate adder can remove its local sum memristor and only requires 3 memristors. This optimization is not performed on the half adders since all but 1 must store their sum bit as part of the final result of the multiplication. In all, N^2-3N+1 sum memristors can be removed. The carry memristor must remain since it and its inverse are sensed differently by the next full adder and do not have a corresponding input memristor to be placed in. The one exception to this rule is the Nth full adder in each ripple carry adder. The carry-out of these full adders serves as the A input to the next full adder, thus it can be stored as an input into the next full adder and the C_{out} memristor removed from the full adder itself. N-3 full adders can remove their carry-out memristors. Also note that this removes memristors from the design but does not remove any switches or drivers. Rather it just moves these switches and drivers onto the next adder's logic. The total savings is N^2-2N-2 memristors.

Lastly, the AND MAD gates can be removed in favor of hybrid-CMOS gates. Hybrid-CMOS AND gates are used because their results can be directly set into the input memristors in the adders. Also, they have a gate delay of 1 and remove N^2 memristors, 3 drivers, and N^2 switches. The resultant schematic for the proposed MAD-based array multiplier is shown in Figure 95.

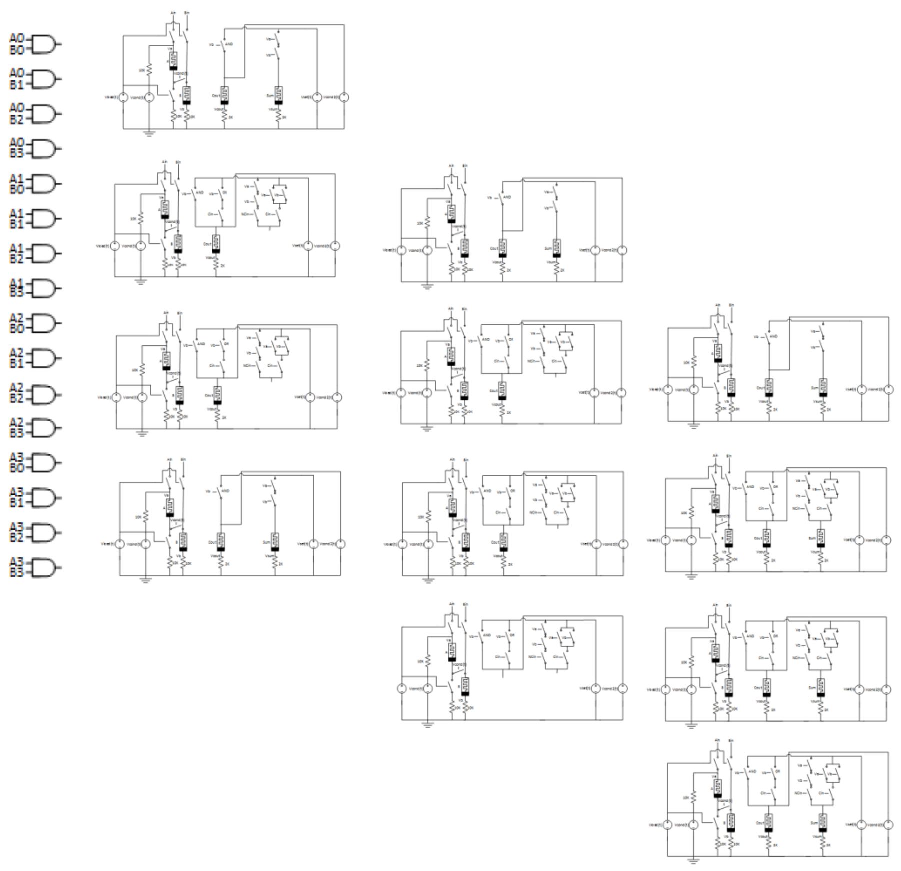


Figure 95: 4-bit MAD array multiplier schematic

The design requires $5N^2 - 2N + 2$ memristors, $3N^2 - 2N - 1$ drivers, and $13N^2 - 19N$ switches. Together, all of the optimizations reduce the delay between each row in the array multiplier by 2 and the initial AND delay by 3. The total delay is only 3N-4 steps for an N-bit multiplication. An example of the resolution of product of 0xF * 0x7 = 0x69 is shown in Figure 96.

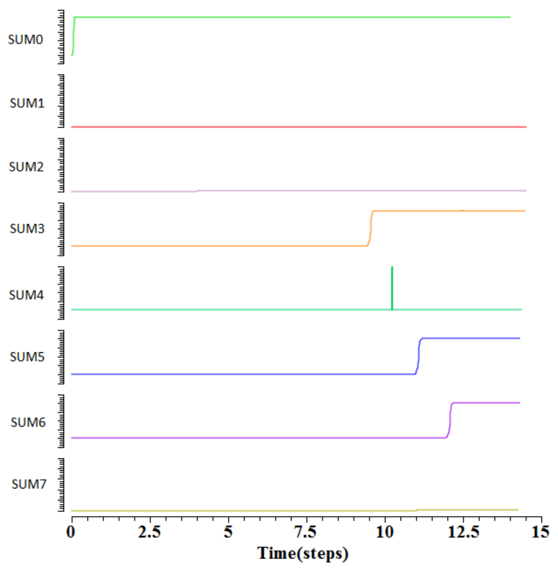


Figure 96: 4-bit MAD array multiplier for 0xF * 0x7 =0x69

Another benefit of this design is its ability to be pipelined. Similar to the other MAD designs, a new multiplication can begin every 4 steps.

ARRAY MULTIPLIER ANALYSIS AND COMPARISON

Table 30 shows the delay and complexity of each of the proposed array multipliers and a traditional CMOS design.

Table 30: Delay and complexity comparisons of the proposed array multipliers

	CMOS	IMPLY	Hybrid-CMOS	Threshold Gates	MAD
Area	$40N^2$-46N MOSFETs	$7N^2$-8N+ 9 memristors $8N^2$-9N+ 9 switches $8N^2$-9N+ 9 drivers	$14N^2$-16N memristors $6N^2$-11N+2 MOSFETs	$9N^2$ -9N memristors $2N^2$-4N GOTOs	$5N^2$ -2N + 2 memristors $13N^2$ - 19N switches $3N^2$ - 2N -1 drivers
Delay	10N-11	24N-35	13N-12	4N-3	3N-4

All of the implementations significantly reduce the number of components as compared to the CMOS design, some to as small as 25%. The IMPLY and MAD implementations require the lowest area. When you consider the overhead of the switches and drivers, the threshold gate implementation requires the fewest components. The hybrid-CMOS implementation requires the most components, including a significant number of MOSFETs. This high amount of CMOS is due to the frequent use of intermediate signal buffering for the carry-out and sum bits.

The threshold-gate and MAD implementations are significantly faster than the CMOS and other approaches, over 2x and 3x faster. The MAD design requires only 3N-4 steps and the threshold-gate implementation requires 4N-3 gate delays. The hybrid requires over 3x the number of steps at 13N-12, slightly higher than CMOS. As expected, the IMPLY operation requires the greatest number of steps, on the order of 24N.

If the delays are pipelined, the IMPLY and MAD designs have improved throughputs. Table 31 shows the delay to perform X consecutive multiplications on the various designs.

Table 31: Delay to perform X consecutive multiplications for the proposed array multipliers

	CMOS	IMPLY	Hybrid-CMOS	Threshold	MAD
Delay	10NX-11X	21X	13NX-12X	4NX-3X	4X

The MAD design is capable of beginning a new multiplication every 4 steps and the IMPLY design every 21 steps. The only consequence is more complicated driver logic, but no additional delay or hardware. The threshold, hybrid-CMOS and CMOS approaches cannot be pipelined so the delay remains X times the delay for a single multiplication.

Dadda Multipliers

Memristor-based Dadda multipliers are presented for the IMPLY, hybrid-CMOS, threshold, and MAD contexts.

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 with a delay of 21 steps and area of 7 memristors. The half adders consist of an XOR and AND operation for a total of 11 steps and 11 memristors. The final stage's addition uses the optimized ripple carry adder in previous sections, requiring 2N+19 steps and 7N+1 memristors. The AND operations are performed in 3 steps with 4 memristors. For a multiplication with S stages and an N-bit final addition, this results in a delay of 21S+2N+22. For an 8-bit multiplication, the delay is 134 IMPLY steps with a complexity of 677 memristors.

The proposed design is optimized to have a delay of 14 IMPLY steps per stage rather than 21 steps for a 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 Dadda multiplier delay has 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.

First, the 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 previous intermediate adders. These inputs are used in a single instance and by a single adder so they can be overwritten.

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 and a half addition and required 24 steps and 15 memristors for two AND operations and a full addition. The proposed design reduces this to 10 steps with 8 memristors and 15 steps with 13 memristors. Table 32 shows the execution steps for performing the AND operation followed by a half addition or full addition.

Table 32: 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 -> NP		B imp 0 -> M0	
2	A imp (B imp 0) -> NP	A NAND B	A imp (B imp 0) -> NO	A NAND B
3	(A imp (B imp 0) imp 0) -> P	A AND B * results now live in P and Q	(A imp (B imp 0) imp 0) -> P	A AND B * results now live in P,Q,R
4	NP imp NQ -> NQ		NP imp NQ -> NQ	
5	Q imp NP -> NP	P NAND Q = NOT COUT	Q imp NP -> NP	P NAND Q
6	P imp Q -> Q		P imp Q -> Q	
7	Reset ALL but Q,NQ, and NP		NQ imp 0 -> P	
8	NQ imp 0 -> P		Q imp P -> P	P XOR Q
9	Q imp P -> P	P XOR Q = SUM	P imp NR -> NR	
10	NP imp 0 -> Ai, Bi	P AND Q = COUT	NR imp 0 -> M1	
11	P imp 0 -> Aj	P XNOR Q = NOT SUM	NP imp M1-> M1	COUT
12			P imp R -> R	
13			R^+ imp P -> P	
14			P imp 0 -> M2	
15			R imp M2 -> M2	SUM
16			M1 imp 0 -> M3	NOT COUT
17			M2 imp 0 -> M4	NOT SUM

Again, A and B represent the inputs to the AND gate, and P and Q represent the inputs to the adder. 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. As shown in Table 32, 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 adder's 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 over delay as an 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. 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 given in Figure 97.

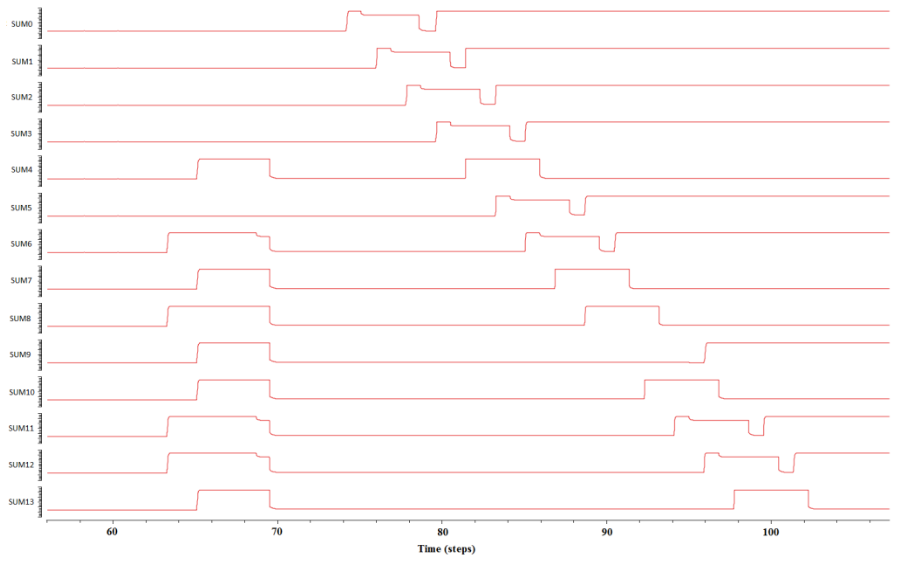


Figure 97: 8-bit IMPLY multiplication execution 0x65*0x43 = 0x1A6F

Although this design does not perform pipelining, pipelining is possible with IMPLY. In order to pipeline this multiplier, each stage could begin a new addition as soon as it completes one and resets its memristors. Thus, a new multiplication can begin every 15 steps.

The area has been reduced as a secondary priority to the delay since IMPLY-based implementations tend to have greater delay than other approaches. The baseline Dadda multiplier had a total complexity of 677 memristors. Numerous area optimizations were performed by reusing memristors within components and across components to reduce this to 385 memristors.

First, note that the half adders do not lie on the critical path for the full adders. Thus, the half adders can reuse memristors by performing reset steps that would otherwise be deemed too costly to the delay. This reduces the area of each half adder by 5 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 operation can be reused immediately and the output memristors can be reused once they are sensed. Again, this saves 4 memristors per input in each of these full adders. Some full adders use intermediate results as one or more of their inputs. In this case, no input memristors are required for the adder since the inputs reside in the result memristors of adders in the previous stage. 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. A schematic for an 8-bit Dadda multiplier with the area optimizations is given in Figure 98.

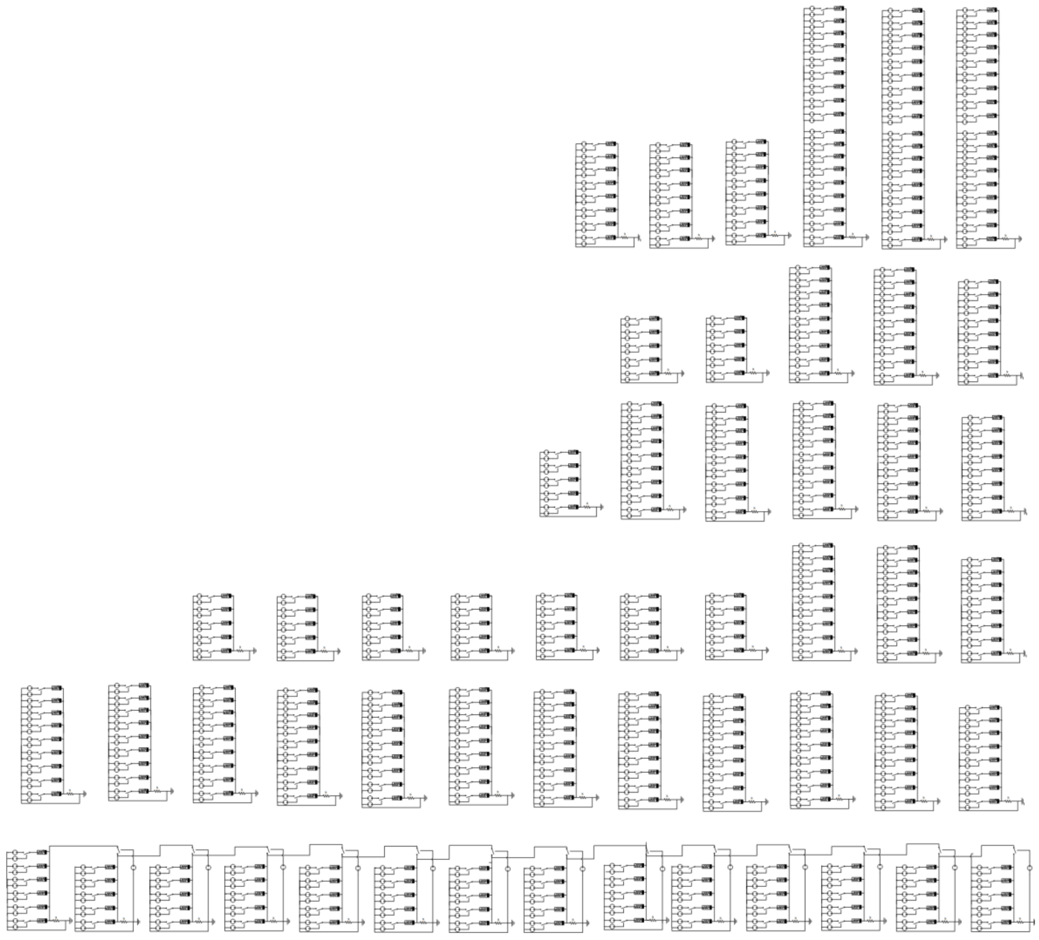


Figure 98: 8-bit IMPLY Dadda multiplier schematic

The stages traverse vertically through the schematic with the final 14-bit ripple carry adder at the bottom row of operation.

HYBRID-CMOS DADDA MULTIPLIER

The hybrid-CMOS approach to an 8-bit Dadda multiplier design has a componential area of 856 memristors and 602 MOSFETs. The Dadda multiplier is especially well-suited for the hybrid-CMOS approach due to its low fan-out. The outputs of the AND gates and the adders all have a maximum fan-out of 2, removing the need for signal restoration. Also, carry propagation buffering is only necessary in the final stage since the intermediate stages use disjoint full adders. Each of these stages requires 6 steps. Signal restoration is still necessary for long series of propagations across stages. Specifically, it was found that for the given design, it was sufficient to buffer the signals after the final two adder stages. The schematic for the hybrid-CMOS Dadda multiplier design is shown in Figure 99.

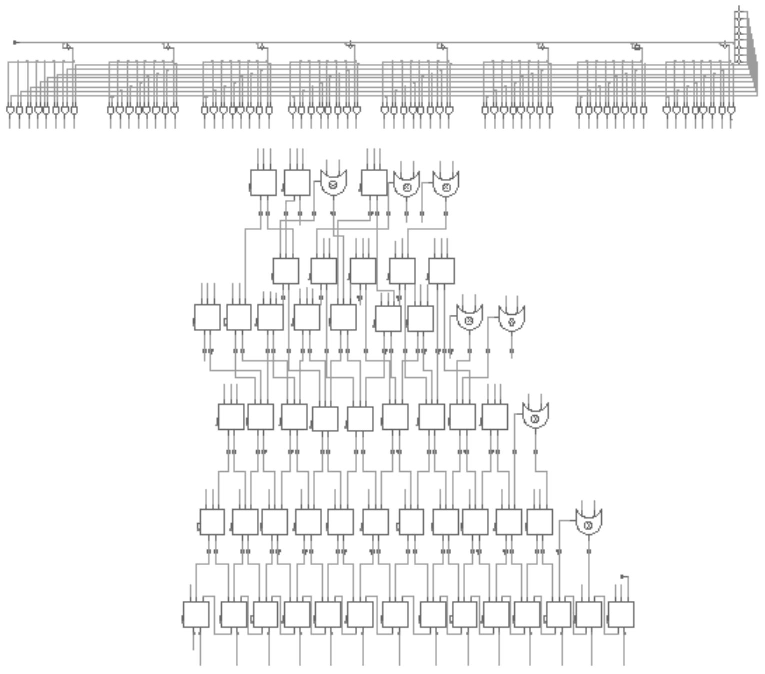


Figure 99: 8-bit hybrid-CMOS Dadda multiplier schematic

The signal restoration adds 84 buffers in addition to the buffers on the carry signals in the final 14-bit adder and the 105 NOT gates in the adders. This results in a componential area of 856 memristors and 602 MOSFETs for the entire design. The signal restoration itself accounts for over 65% of the CMOS in the design. Although this is a high overhead, the design saves area where memristors are used rather than traditional transistors, making the area of the overall design competitive.

In additional to competitive area, the delay of the hybrid-CMOS is low. The initial AND requires 1 step, each intermediate stage requires 6, the final stage requires a buffer step, and the final N-bit addition requires 3N+4 steps. The total delay for a multiplication with S stages is 6S+3N+6. A waveform for the resolution of the 8-bit multiplication 0x65*0x43 = 0x1A6F is shown in Figure 100.

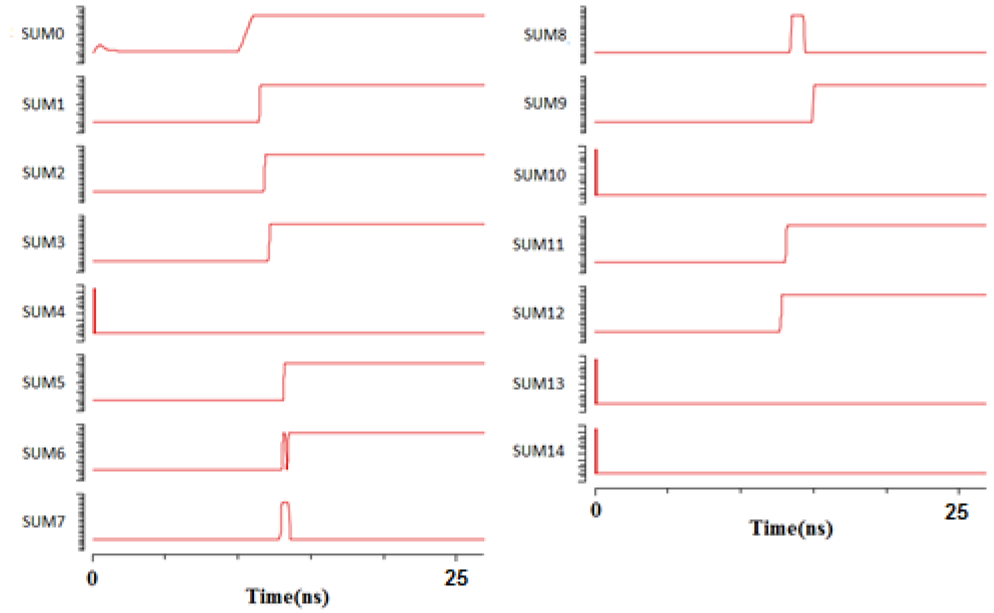


Figure 100: 8-bit hybrid-CMOS Dadda multiplication execution for 0x65*0x43 = 0x1A6F

Even with the CMOS signal buffers, the overall delay is much shorter than the IMPLY approach. The final carry propagation adder accounts for about half of the total delay and the AND gates coupled with the adder stages together account for the second half. If a faster carry propagating adder were used in the final stage, this delay could be further decreased.

Threshold Gate Dadda Multiplier

The threshold gate approach to Dadda multiplier extends the idea of column reduction to leverage the properties of threshold gates. In the traditional Dadda multiplier design, the number of remaining additions is reduced in each stage by using full adders that reduce the height from 3 to 2. Although this approach is competitive, this work presents an alternative implementation that focuses on column reduction but uses gates with more inputs than 3. This approach redesigns the traditional circuitry for Dadda multiplier, removing the final N-bit addition and removing the concept of full adder stages. A schematic for the Dadda multiplier is shown in Figure 101.

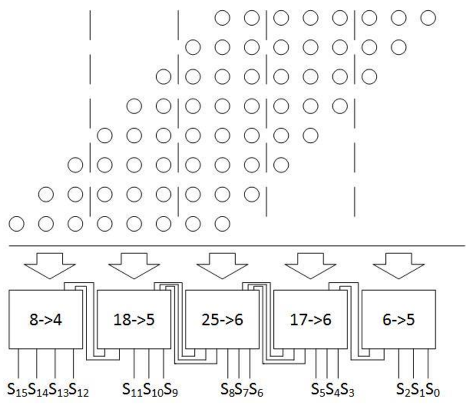


Figure 101: 8-bit threshold gate Dadda multiplier diagram

The 64 AND gates are represented by the circle array. In this figure, counters are used for the main building block in order to achieve improved delay and area as shown previously. The reduction performs on log (columns) of the addition at a time. A logarithmic approach was selected as prior work showed it to balance area and delay. For an 8-bit multiplication, there are 8 rows of 15-bit numbers, so 3 columns are inputs to a single counter. The first 3 columns of the addition feed into a 6->5 counter because the maximum possible value from the addition is 0b10001. The 2 most significant bits of this result are used as inputs along with the next 3 columns of the addition to a 17->6 counter. This pattern continues until the final columns are reached and the final product is resolved. In all, an 8-bit multiplier requires 1 gate delay for the ANDs and 5 counter stages but does not require a final N-bit addition.

Threshold gates are used for the underlying structure of the counters. There are two approaches to threshold gate counter implementations– Kautz and Minnick. Prior work showed the utility of each of these approaches to threshold-gate based counters and multipliers [27]. A summary of the complexity for each of these approaches for standard counter sizes is shown in Table 33.

Table 33: Componential area of threshold-gate counters

	3->2 Area	7->3 Area	15->4 Area
Kautz	7 memristors 2 GOTOs	24 memristors 3 GOTOs	66 memristors 4 GOTOs
Minnick	7 memristors 2 GOTOs	11 memristors 5 GOTOs	26 memristors 10 GOTOs

A diagram showing the construction of a 7->3 counter in each of these methods is given in Figure 102.

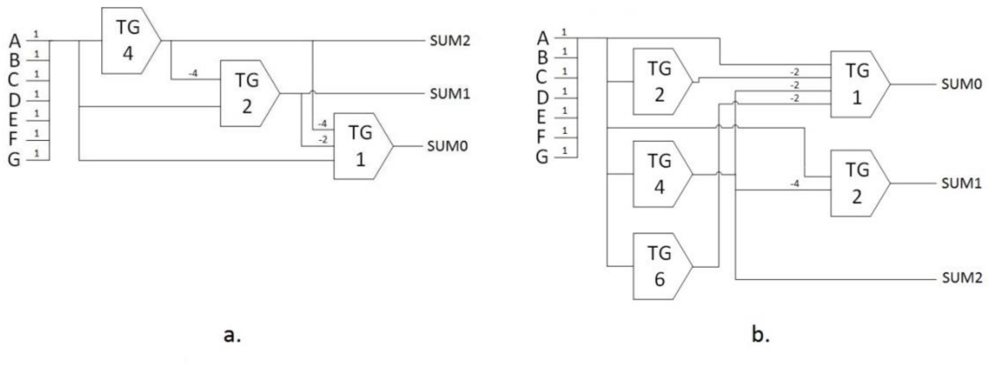


Figure 102: Threshold gate design of a 7->3 counter used a) Kautz and b) Minnick methods

For the Minnick approach, since the same X inputs are used to detect the threshold on the first level of threshold gates, the same X weighted memristors can be used across all gates. Distinct GOTO pairs are still required for each unique threshold detection. Thus, the total number of GOTO pairs is $2^{(q-1)} + q - 2$ and the total number of memristors is $2^q - q - 1 + p$, where p is the number of inputs and q is the number of outputs.

In the Minnick implementation, the most significant bit of the counter output has a single bit delay and the remaining outputs have a delay of 2, regardless of counter width. In the Kautz implementation, the most significant bit of the counter output has a single delay and each subsequent output bit i has a delay 1 greater than the i+1th bit of the output. Both of these delays can be seen in the 7->3 counters in Figure 102.

When these counters are used in the context of an 8-bit Dadda multiplier, each counter C_i must wait to execute until its inputs from counter C_{i-1} are resolved. In total, the proposed Kautz-based approach takes 15 gate delays and a Minnick-based approach takes 11. In terms of area, the 8-bit multiplier requires a 6->5 counter, a 17->6 counter, a 25->6 counter, an 18->5 counter, and an 8->4 counter. For the Kautz approach, this results in only 26 threshold gates in addition to the N^2 AND gates, for a total componential complexity of 731 memristors + 90 GOTO pairs. The Minnick approach requires 174 threshold gates for a total of 379 memristors + 174 GOTO pairs. Table 34 shows the comparison for the proposed two approaches and the traditional approach.

Table 34: Complexity and delay comparisons for two 8-bit threshold gate Dadda multiplier designs

	Kautz	Minnick
Delay	15	11
Complexity	731 memristors 90 GOTO pairs	379 memristors 174 GOTO pairs

Depending on the application and optimization criteria, either the Kautz or Minnick approach should be selected. The Kautz requires about 1/2 of the GOTO pairs but about 2x the number of memristors. If the Minnick method did not share inputs across threshold gates, the Kautz method would have lower area. The Minnick approach is faster, requiring about 25% fewer steps.

MAD GATE DADDA MULTIPLIER

The MAD design for an 8-bit Dadda multiplier has a delay of 20 steps and 298 memristors, 66 drivers, and 697 switches. The design makes many of the same optimizations as the MAD array multiplier. Hybrid-CMOS gates are selected to perform the initial AND operations and the optimized full adders and half adders that require only 4 memristors and 2 steps are used.

Recall that the drivers of these adders are modified to store the sum and carry out results into memristors in the next adders when possible. This is possible for nearly all full and half adders in the design. The only exception is for full adders which do not have an AND gate as an input. For these full adders, at least one of their inputs needs to be stored in a result memristor in the previous adder because it is used as a driver. For the 8-bit Dadda multiplier, 19 of the 42 adders require resident memristors for one of their results. Thus, 23 of the adders require 2 memristors and 19 require 3 for a total of 103 memristors for the intermediate adders.

The delay through each full adder is still a single step. Thus, for an 8-bit multiplication the total delay is S+N+2, or 20 steps for an 8-bit addition. A waveform showing the final product resolution for the 8-bit multiplication 0x65*0x43 = 0x1A6F is given in Figure 103.

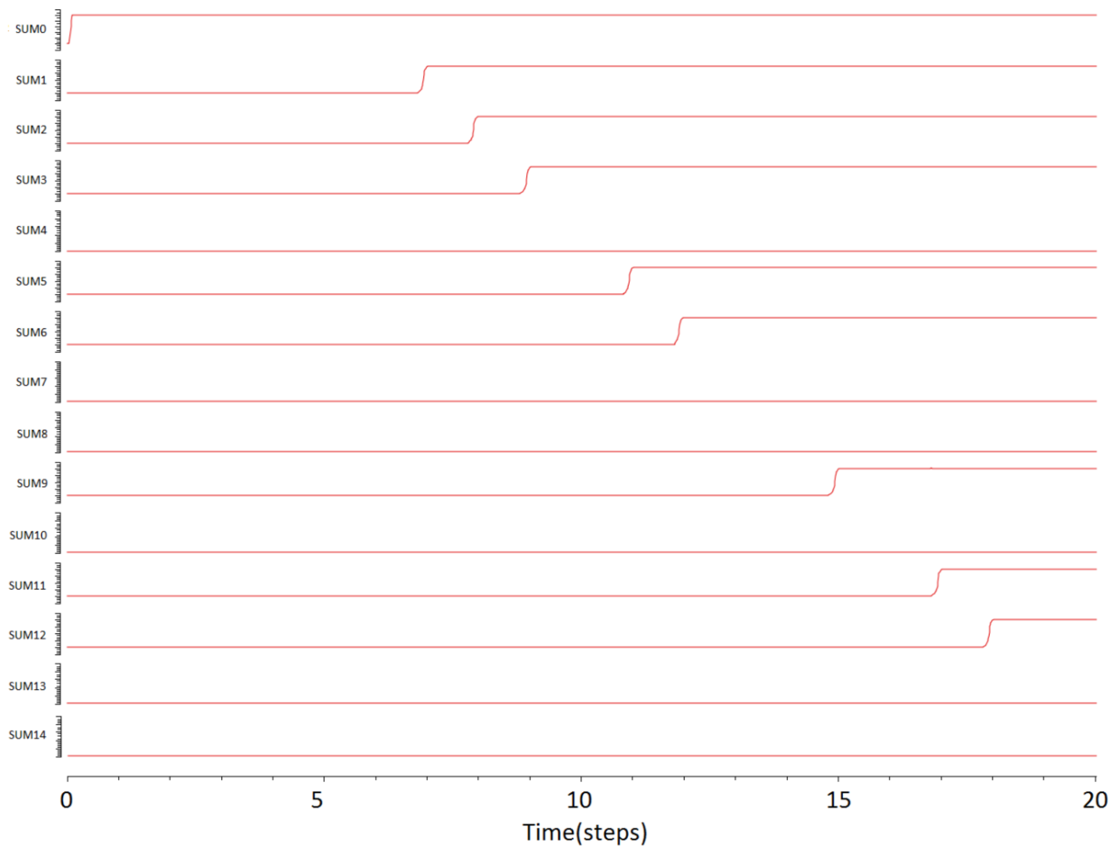


Figure 103: 8-bit MAD Dadda multiplication execution for 0x65*0x43 = 0x1A6F

Note that over 2/3 of the delay is attributed to the final adder. Thus, as the size of the multiplication increases, the benefits of this adder's delay increases. A schematic for the MAD-based approach is shown in Figure 104.

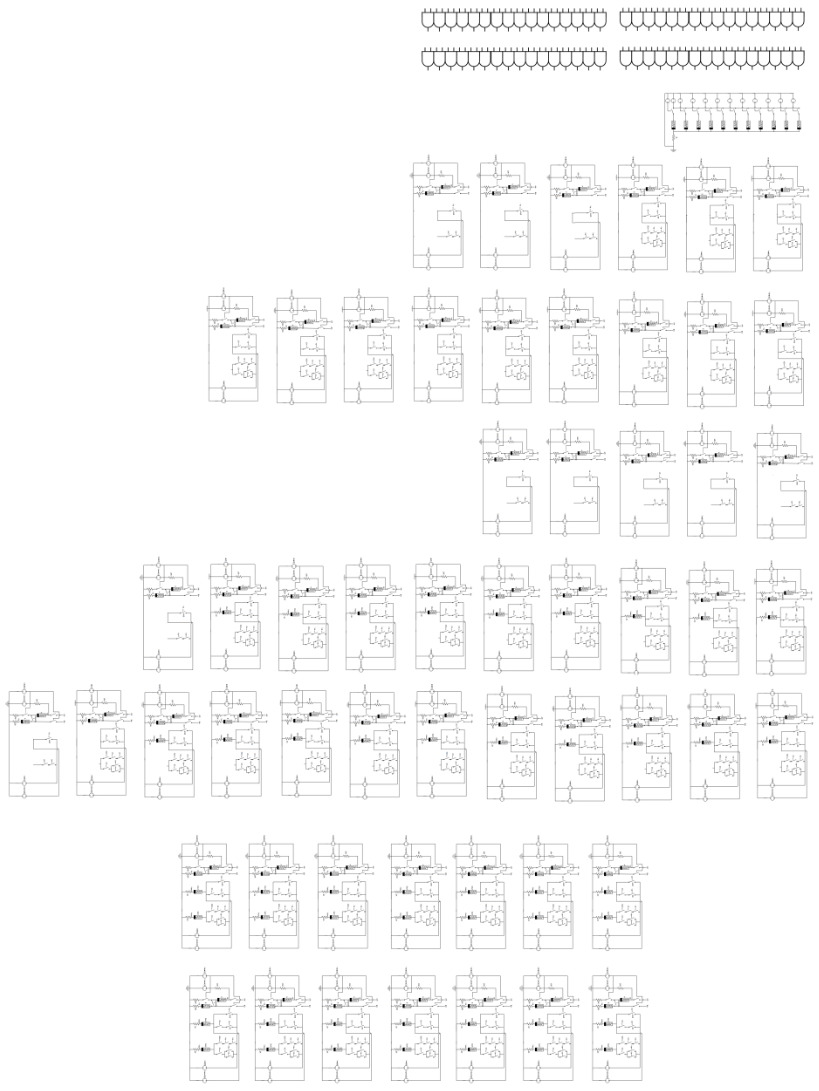


Figure 104: 8-bit MAD-based Dadda multiplier schematic

The hybrid-CMOS AND gates require $2N^2$ memristors, and the final 14-bit ripple carry adder requires 56 memristors. The intermediate full and half adders require a total of 106 memristors. The number of switches for the intermediate adders is 13 per full adder and 7 per half adder for a total of 504 switches. Drivers can be reused across adders executing the same steps so each intermediate stage only requires 1 driver per input since they are potentially loaded in different cycles and 1 for output resolution for a total of 3 drivers per stage. Unfortunately, to maintain bit independence and correct functionality, the same cannot be performed for the switches in the design. There are 11 additional memristors in the first row of the multiplication that are not part of any adders. These memristors hold the partial products A_iB_i that are used later as carry signals into adders. These signals cannot be used as drivers directly from the hybrid-CMOS gates' outputs since the output of the gate should only be sensed and used as a driver during a designated cycle. Thus, the values are stored into intermediate result memristors. In total, the complexity of the design is 298 memristors, 66 drivers, and 697 switches.

The Dadda multiplier can also benefit from pipelining. For example, since execution is based on drivers, after a stage of adders completes its portion of a multiplication, it can begin a subsequent multiplication. Since the bottleneck is a single stage, a new multiplication can begin every cycle.

DADDA MULTIPLIER ANALYSIS AND COMPARISON

Table 354 shows the complexity and delay requirements for each of the presented Dadda multiplier designs. For the threshold approach, the Minnick method is shown since it has a lower delay than the Kautz method and maintains a competitive area.

Table 35: Complexity and delay for the 8-bit Dadda multiplier designs

	CMOS	IMPLY	Hybrid-CMOS	Threshold	MAD
Complexity	2204 MOSFETs	385 memristors 427 drivers 482 switches	856 memristors 602 MOSFETs	379 memristors 174 GOTOs	298 memristors 66 drivers 697 switches
Delay	57	106	72	11	20

The IMPLY, threshold, and MAD-based approaches require the fewest components. The threshold gate approach requires no drivers or switches and has the fewest number of components. The IMPLY approach requires only a few more memristors but requires a significant number of drivers and switches. The MAD approach requires fewer memristors and reduces the number of drivers by 85%. However, it uses 215 more switches than the IMPLY approach. The hybrid-CMOS approach also does not require driver circuitry, but it requires the most memristors with 856. Additionally, the significant use of signal restoration buffers increases the amount of CMOS in the design to 602 MOSFETs, making it the largest of the proposed designs. However, this is still about one-fourth of the number of MOSFETs in the CMOS design and fewer overall components. The other designs further improve the area as compared to CMOS, with threshold implementation requiring 1/4 the number of components.

In terms of delay, each approach has a different time step for its respective delay except the IMPLY and MAD approaches which are equivalent. The MAD-based approach significantly reduces the delay as compared to the IMPLY approach from 106 steps to just 20. This is also almost 1/3 of the CMOS delay. The threshold-gate approach requires the fewest steps at just 11, over a 5x improvement as compared to CMOS. The hybrid-CMOS approach aligns closely with the traditional CMOS delay, with the additional delays for the signal restoration buffers after each intermediate stage.

Table 36 shows the throughput of each proposed design when X consecutive multiplications are performed.

Table 36: Delay comparison of the 8-bit Dadda multiplier designs for X multiplications

	CMOS	IMPLY	Hybrid-CMOS	Threshold	MAD
Delay	57X	15X	72X	11X	X

As described earlier, because of their driver-centric nature, the IMPLY and MAD approaches can pipeline numerous multiplications. After each stage of the multiplication finishes its execution and stores its result into the subsequent adder in the next stage, it can execute its work on the next multiplication. Similarly, the final N-bit addition can be pipelined across bits. The resultant throughput is now just a factor of the slowest stage in the pipeline, 15 steps per multiplication for the IMPLY approach and 1 step per multiplication for the MAD approach. In other words, every cycle, a multiplication can complete in the MAD implementation.

IMPROVED AND OPTIMIZED DIVIDER DESIGNS

This section presents completed work in the design and implementation of memristor-based dividers. IMPLY, hybrid-CMOS, threshold gate, and MAD approaches to binary non-restoring, SRT, and Goldschmidt dividers are described and analyzed in terms of complexity and delay. Completed schematics and simulations are also given. For the discussed designs, let the width of the dividend be N = 8 and the width of the divisor be k = 4 unless otherwise specified.

Binary Non-Restoring Dividers

Memristor-based binary non-restoring dividers are presented for the IMPLY, hybrid-CMOS, threshold, and MAD contexts.

IMPLY BINARY NON-RESTORING DIVIDER

Generally, IMPLY-based designs require lower area than other approaches, making it a prime candidate for a binary non-restoring design since it is generally chosen for its low complexity. The proposed design performs a division using just N+7n+1 memristors and a delay of (N-n+2) (2n+22)-6 steps. Several optimizations were performed to reduce the delay without increasing the area in order to keep area the prime concern.

First, the divisor is copied into the divider to calculate its 2's complement with a NOT operation followed by an ADD. Since an IMPLY copy operation is two consecutive NOTs, the first three steps are NOTs. Since none of the intermediate results of these operations are used again, this reveals two steps of redundancy and 2n unnecessary memristors. Instead, the divisor is NOT'ed directly into the input operand of the adder. This places the operand into the input memristor and performs the first operation of the ADD in a single step rather than 3 and removes the 2n memristors.

Secondly, the carry-out of the adder can be ignored and the final bit of the adder simplified to require 2 fewer steps and 1 fewer memristor. Lastly, since the second operand of the 2's complement addition is known to be 1'b1, the first few steps of operation on it can perform a priori. This results in a total delay of 2n+17, which is 25 steps for a 4-bit divisor.

The waveform for the execution of a 4-bit 2’s complement operation for the input 0b0111 is shown in Figure 105.

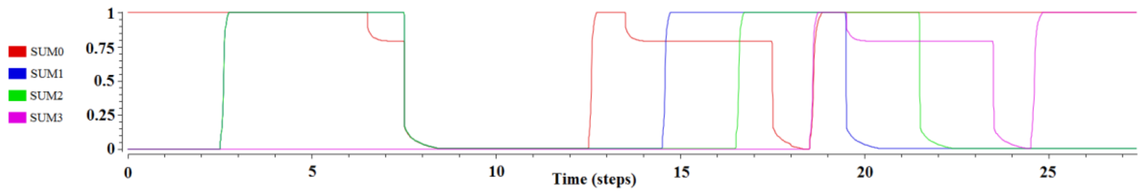


Figure 105: IMPLY 2’s complement operation

The sum bits are resolved in steps 18, 20, 22, and 24 respectively.

More optimizations can be performed given the fact that the 2’s complement of the divisor is used in all subsequent additions in the design. First, all of the additions in a binary non-restoring divider are serialized and have the same width. Thus, the adder circuitry that was used to compute the 2’s complement can be reused. Now, the result of the 2’s complement is stored directly into the input memristors rather than the sum memristors when it is resolved. It stays resident for the remainder of the division since it is used in every addition. This removes copy steps from each iteration and removes the sum memristors.

In the traditional ripple carry adder, the first three steps perform NOT and COPY operations on the input operands. This can be performed as part of the 2’s complement operation so that when each addition iteration begins, the first steps on the 2’s complement input operand have already occurred without the need for additional delay or area. This removes two steps from all subsequent additions. The execution of the first addition for a dividend of 0x77 and divisor of 0x7 is shown in Figure 106.

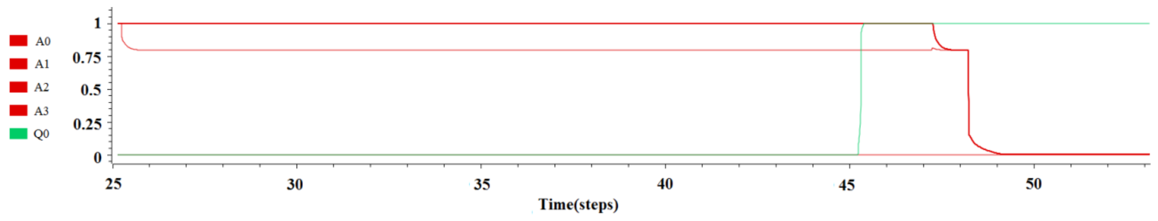


Figure 106: IMPLY divider execution for the first iteration of 0x77 / 0x7

Lastly, both the mux and shift functionality can be incorporated into the adder to remove the need for any additional hardware and reduce the delay by overlapping operations. The shift operation can be performed by storing the result of each full adder into the next full adder rather than itself and shifting in the next dividend bit into bit 0. This performs both the addition and the shift without extra memristors or steps. The divider uses a mux based on the sum of the addition before storing into the operand memristors. If the sum is positive (MSB=0), the sum is selected, else the original dividend operand is. The mux output is then shifted as described for the subsequent addition. Prior work showed that an n-bit mux requires 6 steps and 5n memristors. In this design, the mux is integrated into the adder to reuse intermediate memristors and still achieves the functionality in 6 steps. The steps for this mux are shown in Table 37.

Table 37: IMPLY mux and shift execution steps

Step	**Mux Functionality**	**Mux Goal**
1	(NOT S) IMP 0 -- into ALL bits	S lies in all adders
2	S IMP (NOT Div)	S NAND Div
3	(NOT Sum) IMP 0	Sum
4	Sum IMP S	Sum NAND NOT S
5	(Sum NAND NOT S) IMP 0	Sum AND NOT S
6	(S NAND Div) IMP (Sum AND NOT S)	(S AND Div) OR (NOT S and Sum)

Let S be the MSB of the sum. As part of the addition process, the inverse of the dividend and the inverse of the sum are available since the final store has not yet performed. This also implies the inverse select line is available since it's a sum bit. No extra steps are required to obtain these values. In the first step, the mux performs (NOT S) IMP 0, where the cleared 0 memristor is a reset memristor in each of the full adders. Now, each adder can perform the remaining 5 steps in parallel locally. In step 4, the cleared memristor that the result is stored into is strategically chosen to be the original input memristor. This way, the next addition is ready to begin as usual when the multiplexer completes. The execution of the second addition shows this functionality in Figure 107.

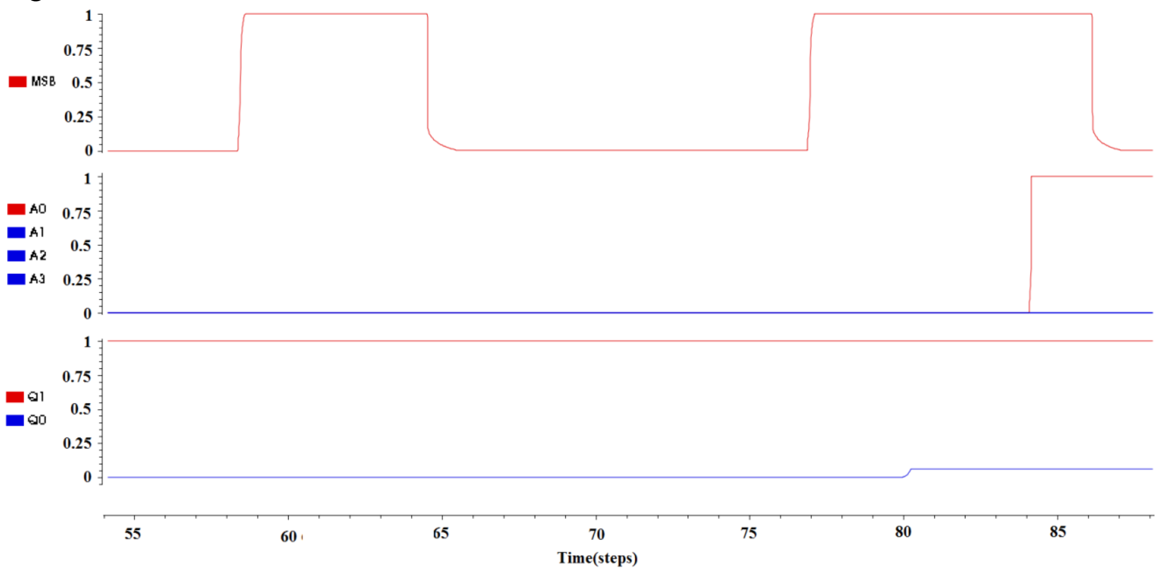


Figure 107: IMPLY divider execution for the second iteration of 0x77 / 0x7

The most significant bit becomes a 1, indicating that the sum was negative and should not update the dividend. Thus, the dividend A remains 3'b000. Then the shift register shifts a 0 in to the 0th spot for the quotient to make the running quotient b10XXX, the dividend's top n bits become 4'b0001, and the most significant bit is reset along with the remaining memristors to 0. Now, the next addition can begin.

The baseline divider implementation with the same optimized individual components but without the contextual optimizations described here requires 2n+19 steps for each addition plus 4 delays for copying each of the input operands in each iteration. Each shift would require 2 steps and each multiplexer would require 6 steps. Thus, the delay of the baseline IMPLY implementation is (N-n+2) (2n+31)-8 steps. For an 8-bit dividend and a 4-bit divisor, this is a total delay of 226 steps. The overall delay of the presented optimized binary restoring divider is (N-n+2) (2n+22)-6 steps, 174 steps for an 8-bit dividend and 4-bit divisor. Nine steps per iteration are removed for a total of 52 fewer steps than the baseline, a near 25% improvement. The 2's complement addition takes 2n+17 steps plus one step to copy in the dividend. The remaining additions combined with the shift and multiplexer functionality take 2n+22 steps. The only exception is the first addition which takes two fewer steps. Note that it is not possible to pipeline divisions since the circuitry is already pipelined for the individual iterations.

The baseline implementation requires 2N+20n+2 memristors. The proposed design requires less than half the number of memristors making it almost the equivalent area of an n-bit IMPLY ripple carry adder. The only additional hardware is N memristors to hold the N-bit dividend and final result. One memristor can be removed from the last bit of the adder because the final carry out is ignored. Thus the total complexity is N+7n-1 memristors plus their drivers. A schematic for an 8-bit by 4-bit divider is shown in Figure 108.

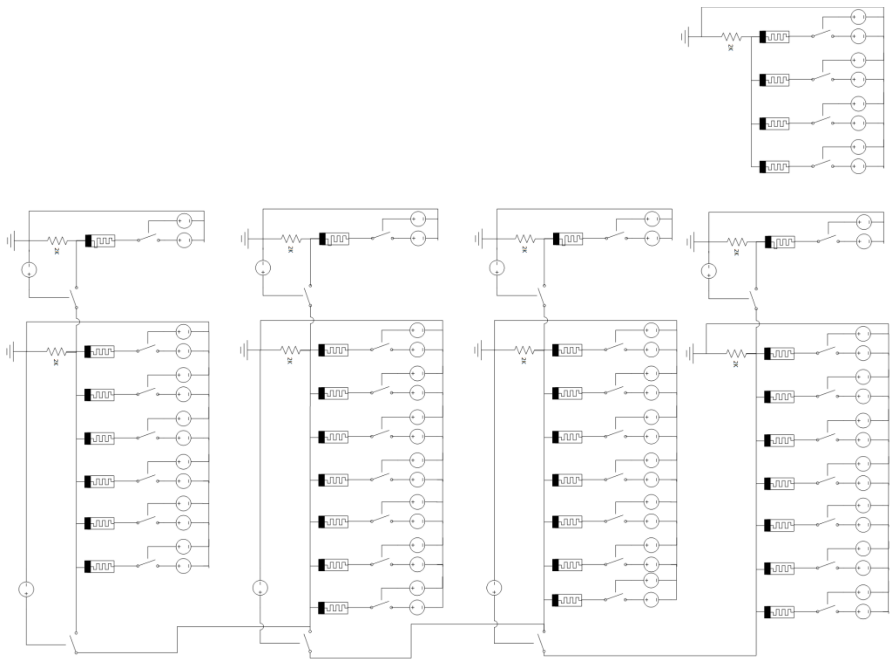


Figure 108: IMPLY binary non-restoring divider schematic

HYBRID-CMOS BINARY NON-RESTORING DIVIDER

The low fan-out of the signals in the binary non-restoring divider make hybrid-CMOS gates a practical option for a design because minimal signal restoration buffering is needed. The proposed design also minimizes the need for these signal restoration buffers by placing them in at optimal granularity in the design. The schematic for the proposed hybrid-CMOS binary non-restoring divider is shown in Figure 109.

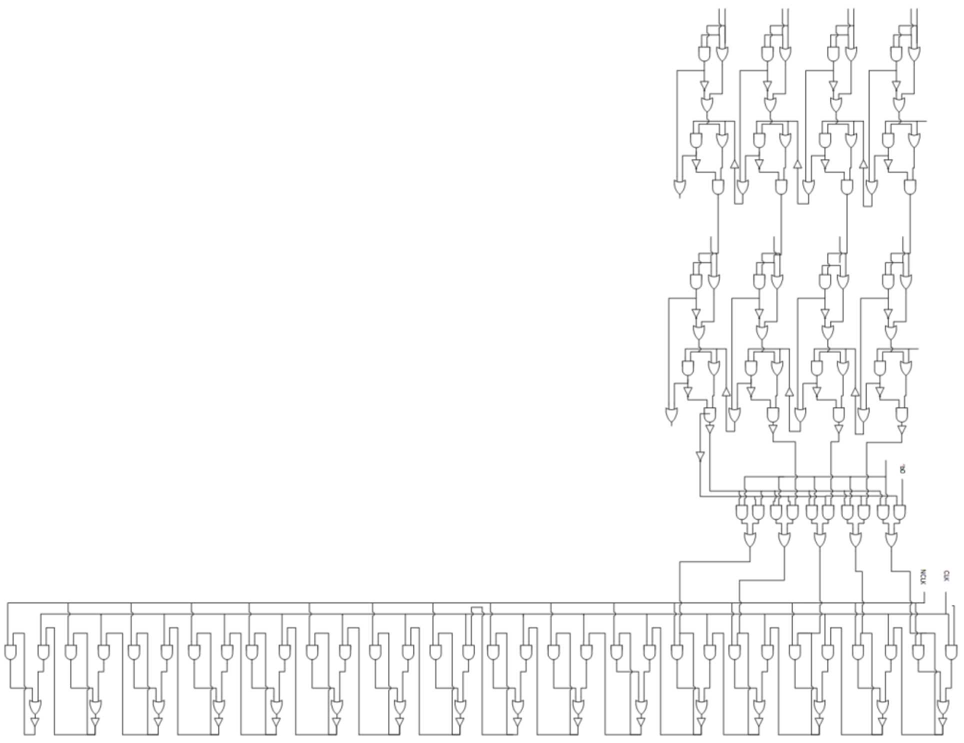


Figure 109: Hybrid-CMOS binary non-restoring divider schematic

The design consists of 2 n-bit adders, an n-bit mux, and an N-bit shift register. The adders each require 14n memristors and 8n MOSFETS and the multiplexer requires 6n memristors and 2 MOSFETs. The adder hardware cannot be reduced in the same way as the IMPLY design due its combinatorial nature. The shift register requires 12N memristors. As shown, the shift register also requires signal restoration buffers at the outputs of each latch due to the feedback behavior. This adds 2N NOT gates (4N MOSFETs) to the design. Also, the outputs of the 2nd adder must be buffered due to the series of gates in the mux and shift logic that the signals must propagate through before reaching the shift register. This adds another 2n NOT gates, or 4n MOSFETs, for a total CMOS count of 4(N+n) MOSFETs. The total complexity for the proposed 8-bit by 4-bit design is 34n+12N+6 memristors and 4N+24n+2 MOSFETs.

The gate delay of the design is equivalent to the CMOS design plus an additional gate delay at the output of the addition and at the output of the latches in the shift register for buffering so the ripple carry adder takes 3n+5 cycles and the shift register takes 3. The mux only takes 2 cycles instead of 3 because the select line resolves in an earlier cycle. Thus, the total delay is (N+n-2)(3n+10)-6. A waveform for initialization and the first iteration of the division 0x77 / 0x7 is shown in Figure 110.

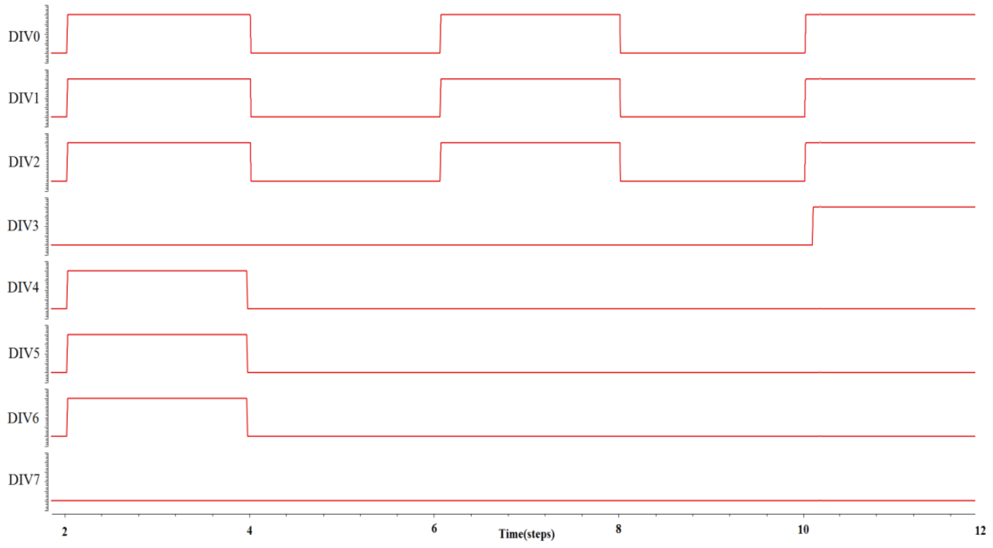


Figure 110: Hybrid-CMOS binary non-restoring divider execution for 0x77 / 0x7

In the first step, the dividend, 0x77 is loaded into the shift register. Then the first addition completes with a result of 0x07 loaded into the shift register. Lastly, the shift register shifts and the quotient bit shifts in to bit 0 for a result of 0xF. Now the next iteration can begin.

THRESHOLD GATE BINARY NON-RESTORING DIVIDER

The design of a binary non-restoring divider using threshold gates requires 10N+26n+2 memristors + 4N+6n GOTO pairs + 6 MOSFETs and a delay of (N-n+2)(n+4). The n-bit ripple carry adder requires n+1 gate delays and 7n memristors and 2n GOTO pairs. This work proposes an implementation of the remaining hardware which combines the multiplexer and shift register into a single structure to reduce complexity and delay. Figure 111 shows a diagram for the various types of cells in this structure.

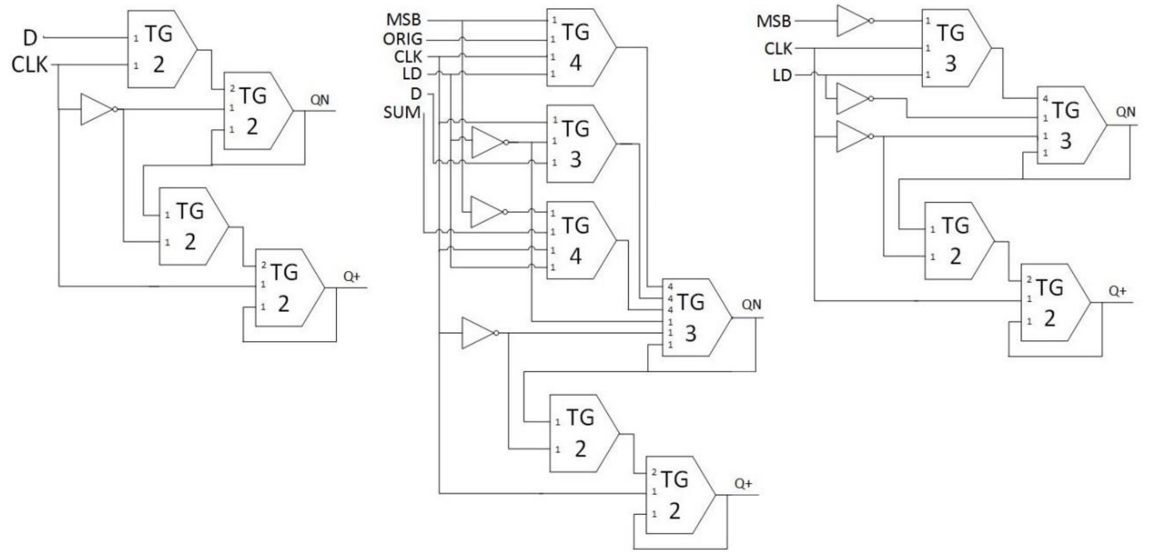


Figure 111: Various cells for the shift register and multiplexer in the threshold gate based binary non-restoring divider

There are three types of cells: standard flip-flops, flip-flops with a load capability from the addition, and flip-flops with a load capability of the quotient specifically. Note that the third cell can be implemented using the second cell, but this work simplifies the cell since the inputs are known. The first cell is standard and requires 2 MOSFETs, 4 GOTO pairs, 10 memristors, and a delay of 2. The second cell has a load capability which selects either the current shift register value or the new sum, dependent on whether the result of the subtraction was positive or negative. This can be seen with the three top-most threshold gates. The remaining gates behave the same as in the standard cell. The third cell simply shifts a 1 or 0 into the shift register based on the MSB of the result. This is seen in the top-most threshold gate. The shift register and mux require (N-n-1) of the first cell, n of the second cell, and 1 of the third cell. Note that the inputs and outputs on the NOT gates are the same three signals across all three cells. Thus, the design only requires a total of 3 NOT gates or 6 MOSFETs to produce all necessary input signals. In all, the N-bit shift register/multiplexer requires 10N+12n+2 memristors + 4N+2n GOTO pairs + 6 MOSFETs. The two n-bit adders add 14n memristors and 4n GOTO pairs for a total componential complexity of 10N+26n+2 memristors + 4N+6n GOTO pairs + 6 MOSFETs.

In terms of delay, the critical path consists of the addition, the multiplexer, and the shift. The delay for the addition is n+1 delays, and the longest delay for the shift-register and multiplexer cells is 3 steps, for a total critical path of n+4 delays. The execution requires the initial 2's complement addition followed by N-n+1 addition iterations to perform the division. Thus, the total delay is (N-n+2) (n+4).

MAD GATE BINARY NON-RESTORING DIVIDER

The MAD binary non-restoring divider has an complexity of N+4n memristors, N+3n+1 drivers, and a delay of (n+2)(N-n+2) steps. The schematic is shown in Figure 112.

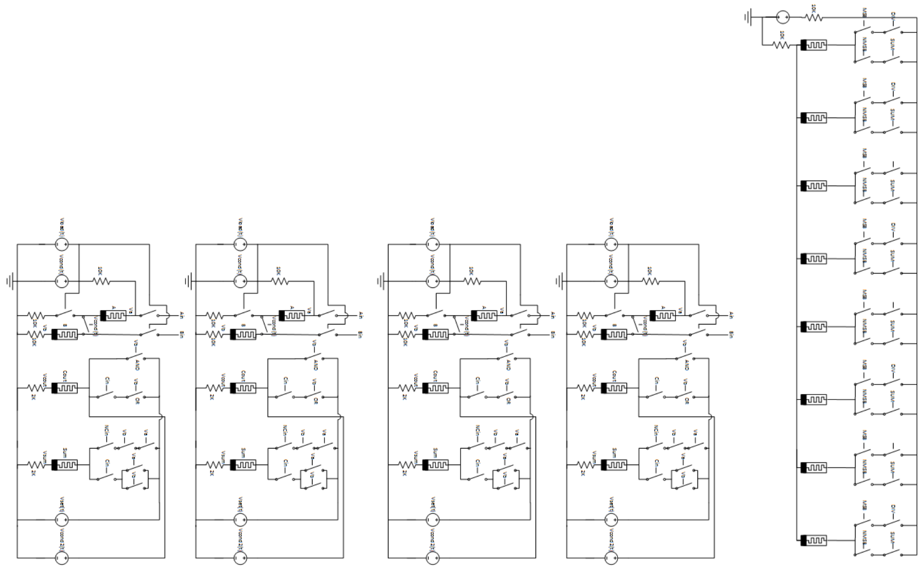


Figure 112: MAD binary non-restoring divider schematic

The design only requires a single n-bit ripple carry adder and an N-bit register. The N-bit register holds the current dividend. The multiplexer and shift functionality are incorporated into the adder as described for previous designs. The driver circuitry must change slightly from the standard ripple carry adder to accommodate these optimizations. First, it is adapted to shift its result by storing the sum in each bit i into the input operand A in bit i+1 instead as described for previous designs. However, the adder must determine the final sum bit before resolving the sum into the input memristors in any of the bits.

In step n+1, the final full adder bit in the ripple carry adder executes and resolves its sum. This sum in turn can drive the resolution of the remaining sum bits. Before this can occur, the input memristors A must be cleared. Thus, all of the bits that have already completed their additions clear their input memristor A in step n+1 while the final bit executes. In step n+2, if the MSB of the sum is 0, each input A memristor and the dividend register are cleared. Else, nothing changes. In step n+3, the shift and first "load" step of the next iteration is executed. If the most significant bit has a sum of 0, the drivers drive the result of the sum into both the A input memristors and the dividend register. For example, the drivers for bit 3 will drive the new sum value from bit 2.

The total delay is (N-n+2) (n+2) steps. The waveform for the first two iterations of the division 0x77 / 0x7 is shown in Figure 113.

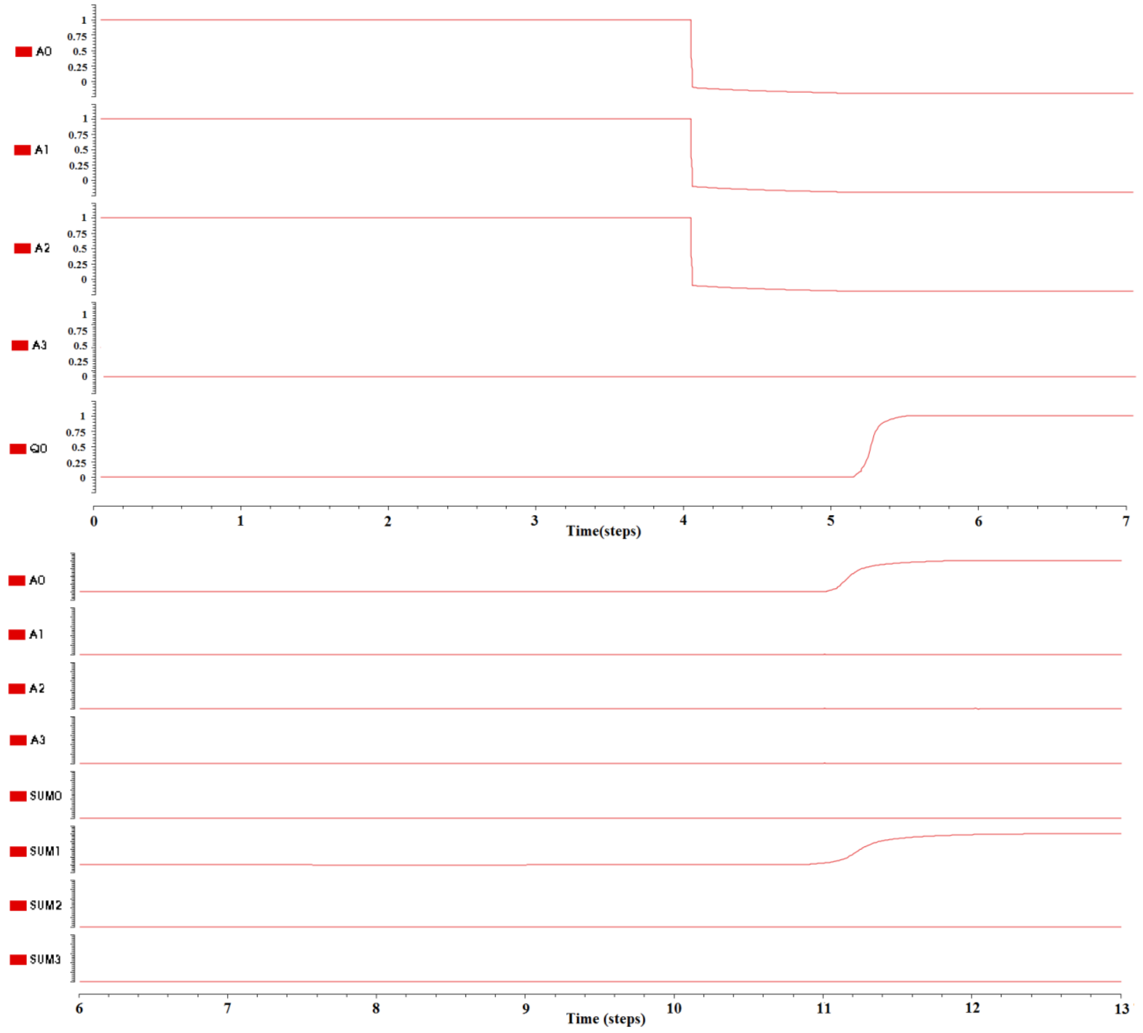


Figure 113: MAD binary non-restoring divider execution for the first two iterations of 0x77 / 0x7

In step 4, the A input memristors in each bit are cleared. In step 6, the driver to set the sum goes high and the A memristors resolve to the updated sum b000 and the quotient bit goes high. In the second iteration, the resultant sum is negative. Thus, rather than accept the shifted sum value of b0010, the adder drives the original dividend with the value b0001. The quotient is now updated to hold b10 in step 12. Note that it is not possible to pipeline divisions since the circuitry is already pipelined for the individual iterations.

BINARY NON-RESTORING DIVIDER ANALYSIS AND COMPARISON

Table 38 summarizes the complexity and delay of each of the proposed binary non-restoring dividers as well as the CMOS design.

Table 38: Complexity and delay comparisons for the proposed binary non-restoring divider designs

	CMOS	IMPLY	Hybrid-CMOS	Threshold	MAD
Delay	(N-n+2)(2n+9)-5	(N-n+2)(2n+22)-6	(N-n+2)(3n+10)-6	(N-n+2)(n+4)	(N-n+2)(n+2)
Area	36N + 86n + 22 MOSFETs	N+7n-1 memristors N+8n-1 drivers	34n+12N +6 memristors 4N+24n+2 MOSFETs	10N+26n+2 memristors 6 MOSFETs 4N+6n GOTO pairs	N+4n memristors N+3n+1 drivers

In terms of delay, the IMPLY and hybrid-CMOS approaches have roughly equivalent delays and the threshold and MAD approaches have roughly equivalent gate delays. The IMPLY and hybrid-CMOS designs have about double the delay of the threshold and MAD based designs. However, the delays are still competitive with the CMOS. The proposed MAD-based design has the shortest delay of (N-n+2) (n+2). This means that each iteration of the subtraction only takes n+2 cycles, less than half of CMOS.

The IMPLY and MAD-based approaches require the fewest components, with MAD requiring only N+4n memristors and their accompanying drivers. This is less than

10% of the number of components in the CMOS design and the components themselves are smaller. The MAD-based approach requires only 2n more drivers in order to achieve its lower delay. The threshold gate implementation requires the most components, on the order of 14N as compared to 2N, in order to achieve its low delay. However, this design keeps the use of CMOS very low with just 6 MOSFETs. The amount of CMOS in the hybrid-CMOS implementation is much greater, requiring 4N+24n+2 MOSFETs for the design however this is a significant reduction compared to CMOS. Both the threshold and hybrid-CMOS approaches have roughly the same number of nanoscale components depending on the values of N and n.

SRT Dividers

This section presents four approaches to SRT dividers constructed from memristors. First, an IMPLY approach is implemented and optimized, followed by a hybrid-CMOS, threshold gate, and MAD gate implementation. Each of the designs is compared and analyzed against the traditional CMOS design.

IMPLY SRT DIVIDER

Let the SRT divider perform X iterations before converging. Based on prior work, 2k+19 cycles are required for the initial 2's complement, 2 cycles are required to copy the result, 2 cycles are required to copy the dividend, and two cycles are requires to shift the result. Then, the mux requires 38k+2 steps for the serialized k-bit 4-to-1 mux followed by 2k+19 steps for the next addition. For X iterations, the total delay of the baseline implementation is 2k+25+(X+1) (40k+21). The baseline's total area is comprised of the 2's complement adder, the iterative divider, the multiplexer, and the correction logic. This totals to 87(k+1)-2 memristors and their drivers. This work performs optimizations to this baseline IMPLY approach to reduce the complexity to 20% to 16(k+1)-2 memristors and 18n+4 drivers with a delay of (X+2) (2k+25)-5.

First, the 2's complement of the divisor is computed. Previous sections showed that this can be performed in 2(k+1) +17 steps with 7(k+1)-1 memristors. For this design, a copy of the initial divisor and 2's complement must be retained for use in future iterations, requiring 2k more memristors. These COPY operations can be incorporated into the 2's complement operation itself so that they do not incur any delay penalty.

In order to reduce area, the adder that was used to compute the 2's complement is reused for the remaining iterations of execution, removing 7(k+1) memristors from the design. This is possible because each iteration is serialized and of equal size k+1. The same optimizations performed for other proposed designs to incorporate the mux and shift functionality into the adder are performed. The result of the sum is stored into the input A memristors so the dividend comparison can be performed in the subsequent step. This optimization removes 2 steps and n memristors. An example of this execution for the first addition iteration of the SRT division 0.0101 by 0.1100 is shown in Figure 114.

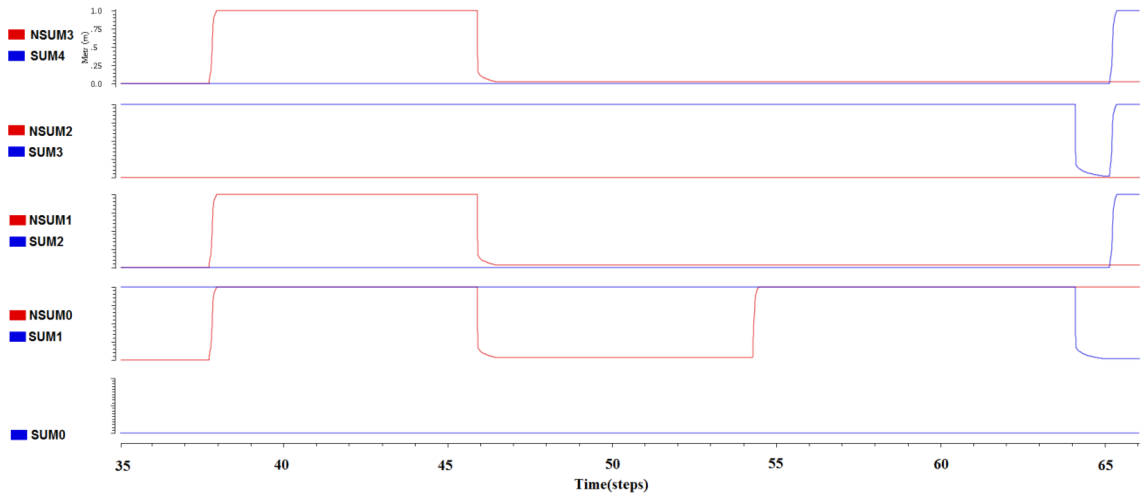


Figure 114: Execution of the first addition of the SRT division 0.0101 / 0.1100

The inverse of the sum is resolved as b0.001. At the completion of the sum, each of these values is inverted and stored into its corresponding left-shifted bit to produce 1.1100, the correct shifted sum of 0b0.1010 + 1.0100.

The 4-to-1 mux is optimized in the IMPLY domain to reduce the number of steps by overlapping the steps of the multiplexer with the previous and subsequent additions. A baseline implementation for a 4-to-1 mux based on IMPLY operations requires 38 steps and 41 memristors, not including inputs. A breakdown of the execution steps for the baseline implementation is given in Table 39.

Table 39: Baseline implementation of a 4-to-1 mux for inputs A,B,C,D and select lines Q_{k-2} and Q_{k-1}

Step	Operation	Functionality
1	Set Q_{k-1}	
2	Sense Q_{k-1}	Copy Q_{k-1}
3	Set Q_{k-2}	
4	Sense Q_{k-2}	Copy Q_{k-2}
5	Q_{k-2} imp 0	NOT Q_{k-2}
6	Q_{k-1} imp 0	NOT Q_{k-1}
7	(Q_{k-2} imp 0) imp 0	
8	(Q_{k-1} imp 0) imp ((Q_{k-2} imp 0) imp 0)	
9	(Q_{k-1} imp 0) imp ((Q_{k-2} imp 0) imp 0) imp 0 -> M0	NOT Q_{k-2} AND NOT Q_{k-1}
10	M0 imp 0	
11	A imp (M0 imp 0)	
12	(A imp (M0 imp 0)) imp 0 -> S0	NOT Q_{k-2} AND NOT Q_{k-1} AND A
13	Q_{k-1} imp 0	NOT Q_{k-1}
14	Q_{k-2} imp 0	
15	(Q_{k-1} imp 0) imp (Q_{k-2} imp 0)	
16	((Q_{k-1} imp 0) imp (Q_{k-2} imp 0)) imp 0 -> M1	Q_{k-2} AND NOT Q_{k-1}
17	M1 imp 0	
18	B imp (M1 imp 0)	
19	(B imp (M1 imp 0)) imp 0 -> S1	Q_{k-2} AND NOT Q_{k-1} AND B
20	Q_{k-2} imp 0	NOT Q_{k-2}
21	(Q_{k-2} imp 0) imp 0	
22	Q_{k-1} imp ((Q_{n-2} imp 0) imp 0)	
23	(Q_{k-1} imp ((Q_{n-2} imp 0) imp 0)) imp 0 -> M2	NOT Q_{k-2} AND Q_{k-1}
24	M2 imp 0	
25	C imp (M2 imp 0)	
26	(C imp (M2 imp 0)) imp 0 - >S2	NOTQ_{k-2} AND Q_{k-1} AND C
27	Q_{k-2} imp 0	NOT Q_{k-2}
28	Q_{k-1} imp (Q_{k-2} imp 0)	
29	(Q_{k-1} imp (Q_{k-2} imp 0)) imp 0 -> M3	Q_{k-2} AND Q_{k-1}
30	M3 imp 0	
31	D imp (M3 imp 0)	
32	(D imp (M3 imp 0)) imp 0 - >S3	Q_{k-2} AND Q_{k-1} AND D
33	S3 imp 0	
34	(S3 imp 0) imp S2	S3 or S2
35	S1 imp 0	
36	(S1 imp 0) imp S0	S1 or S0
37	((S1 imp 0) imp S0) imp 0	
38	(((S1 imp 0) imp S0) imp 0) imp ((S3 imp 0) imp S2)	S3 or S2 or S1 or S0 = RESULT

This work proposes an optimized implementation for a 4-to-1 mux that requires only 18 steps and 11 memristors. This reduces the number of steps to less than half of the baseline and reduces the memristors to almost 25% of the baseline. The execution steps for this multiplexer are given in Table 40.

Table 40: Optimized IMPLY implementation of a 4-to-1 mux for inputs A,B,C,D and select lines Q_{k-2} and Q_{k-1}

Step	Operation	Functionality
1	Set Q_{k-1}	NOT Q_{k-1}
2	Sense Q_{k-1}	Copy Q_{k-1}
3	Set Q_{k-2}	NOT Q_{k-2}
4	Sense Q_{k-2}	Copy Q_{k-2}
5	(NOT Q_{k-1}) imp Q_{k-2}	
6	A imp ((NOT Q_{k-1}) imp $Q_{k-2)}$ -> S0	NOT Q_{k-2} AND NOT Q_{k-1} AND A
7	(NOT Q_{k-1}) imp (NOT Q_{k-2})	
8	B imp ((NOT Q_{k-1}) imp (NOT Q_{k-2})) -> S1	Q_{k-2} AND NOT Q_{k-1} AND B
9	(Q_{k-2}) imp (Q_{k-1})	
10	C imp ((Q_{k-2}) imp (Q_{k-1}))-> S2	NOTQ_{k-2} AND Q_{k-1} AND C
11	Q_{k-2} imp (NOT Q_{k-1})	
12	D imp (Q_{k-2} imp (NOT Q_{k-1})) -> S3	Q_{k-2} AND Q_{k-1} AND D
13	S3 imp 0	
14	S1 imp 0	
15	(S3 imp 0) imp S2	
16	(S1 imp 0) imp S0	
17	((S3 imp 0) imp S2) imp 0	
18	(((S3 imp 0) imp S2) imp 0) imp ((S1 imp 0) imp S0)	S3 or S2 or S1 or S0 = RESULT

Two of the inputs to the mux are the divisor and its 2's complement and the other two inputs are constant 0. This further reduces the Boolean equation for the mux to only require 6 steps and 3 memristors. The execution steps for this multiplexer are given in Table 41.

Table 41: Optimized IMPLY implementation of a 4-to-1 mux for SRT

Step	Operation	Functionality
1	Q_{k-1} imp Q_{k-2}	
2	Q_{k-2} imp Q_{k-1}	
3	D_{k-1} imp Q_{k-2}	NAND(Q_{k-1} NQ_{k-2} D_{k-1})
4	ND_{k-1} imp Q_{k-1}	NAND(NQ_{k-1} Q_{k-2} ND_{k-1})
5	(D_{k-1} imp $Q_{k-2)}$ imp 0 -> B	Store operand into adder
6	Q_{k-1} imp B	Resolve operand value

Note that the result will be stored back into the adder into the B input memristors. This reduces the complexity of the multiplexer from 3(k+1) to 2(k+1) and removes the 2 load steps from the baseline implementation.

In order to extend this 4-to-1 mux to a width of k and enable parallelizability, each bit must have an independent memristor for both Q_{k-1} and Q_{k-2} since the mux operation overwrites these values. Thus, when these bits of the new dividend are resolved and stored back into the adder, they should also be loaded into these memristors. This requires 2(k+1) additional memristors and 2 drivers to the design but does not introduce any additional delay. Thus, the complexity for the 2's complement, adder iterations, and multiplexer logic is (X+1) (2k+25)-5 steps and 11(k+1)-1 memristors and 12(k+1) +2 drivers. This reduces the delay of the baseline by 20x, from 40kX to just 2kX while also reducing the componential count by 75% from 55k-2 to 11k-1 memristors.

Lastly, the logic for converting the quotient from binary signed digit to binary and correcting both the remainder and quotient if required is introduced. The conversion can be performed on-the-fly during each iteration j of the division by calculating the values of Q[j] and QM[j] as described earlier in this work. This conversion also performs the correction based on the remainder. The equations for Q[j] and QM[j] can be performed using IMPLY operations as shown in Table 42a and 42b.

Tables 42a: Execution steps for calculating Q[j] and QM[j] for iteration j

	OPERATION for Q[j]	FUNCTIONALITY	OPERATION for QM[j]	FUNCTIONALITY
1	Q_{k-1} imp Q_{k-2} ->M0		Q_{k-2} imp Q_{k-1} -> M2	
2	M0 imp 0 -> M1		M2 imp 0 -> M3	
3	QM[j] imp M0	QM[j] NAND (Q_{k-1} AND Q_{k-2})	Q[j] imp M2	Q[j] NAND (Q_{k-2} AND Q_{k-1})
4	Q[j] imp M1	Q[j] NAND (Q_{k-1} NAND Q_{k-2})	QM[j] imp M3	QM[j] NAND (Q_{k-2} NAND Q_{k-1})
5	FALSE $Q[j]_{n:1}$	RESET $Q[j]_{k:1}$	FALSE $QM[j]_{k:1}$	RESET $QM[j]_{k:1}$
6	M1 imp 0 -> NEW $Q[j]_{k:1}$		M3 imp 0 -> NEW $QM[j]_{k:1}$	
7	M0 imp NEW $Q[j]_{n:1}$	(QM[j] AND Q_{k-1} AND Q_{k-2}) OR (Q[j] AND (Q_{k-1} NAND Q_{k-2}))	M2 imp NEW $QM[j]_{k:1}$	(Q[j] AND Q_{k-2} AND Q_{k-1}) OR (QM[j] AND (Q_{k-2} NAND Q_{k-1}))

Table 42b: Execution steps for calculating $Q[j]_0$ and $QM[j]_0$ for iteration j

STEP	OPERATION for Q[j]	FUNCTIONALITY
1	Q_{k-1} imp Q_{k-2} ->M0	
2	M0 imp 0 -> $Q[j]_0$	
3	Q_{k-2} imp Q_{k-1} -> M1	
4	M1 imp $Q[j]_0$	NEW $Q[j]_0$
5	$Q[j]_0$ imp $QM[j]_0$	NEW $QM[j]_0$

The resolution of the most significant k-1 bits takes 7 steps and the lowermost bit of QM[j] and Q[j] takes only 5 steps. Also, since the values of Q_{k-1} and Q_{k-2} are already set and sensed as part of the division iteration, these values are available for each bit of Q[j] and QM[j] to allow each bit to resolve in parallel. A waveform showing the final resolution of the Q[j] and QM[j] values for the SRT division 0.0101 by 0.1100 is shown in Figure 115.

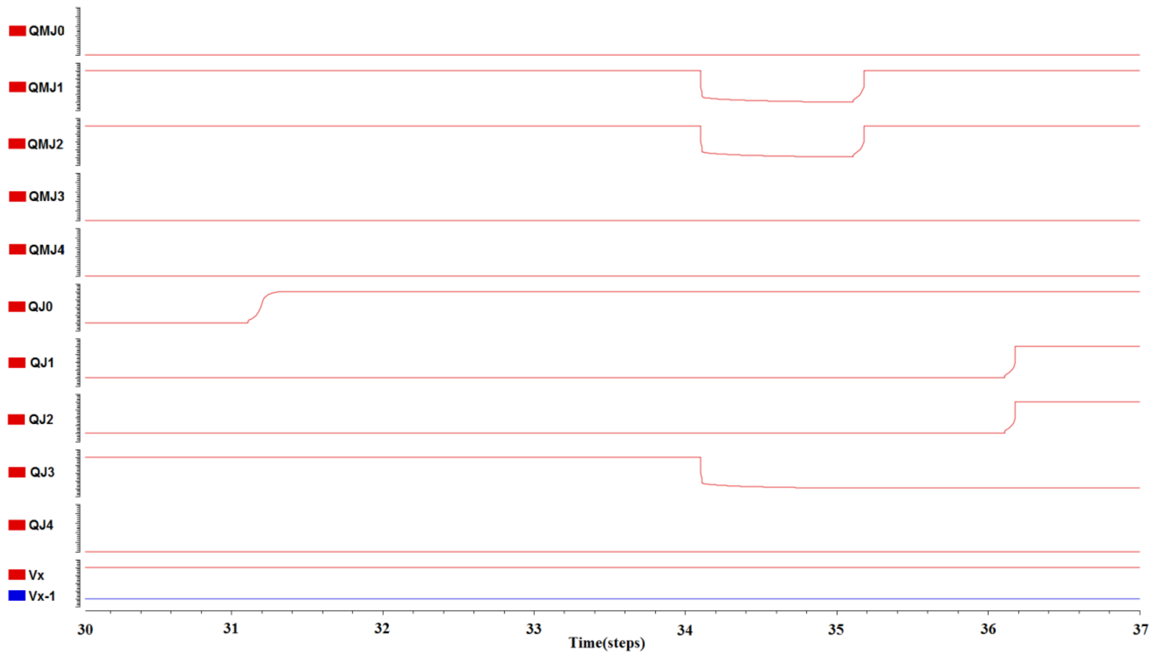


Figure 115: Execution of Q[j] and QM[j] for 0.0101 / 0.1100

The multiplexer correctly identifies that the result of the addition is less than 0.5, so Q[j] resolves to 0.0111 and QM[j] resolves to 0.0110, the final quotient.

The entire operation requires only 4k-2 intermediate memristors, 4 extra drivers, and 2k memristors for Q[j] and QM[j]. The design strategically selects the order of IMPLY operations such that the Q[j] and QM[j] registers are only read and never written. Instead, once the pertinent information is stored in the intermediate registers M0, M1, M2, and M3, both Q[j] and QM[j] can be reset and rewritten as done in steps 5-7 in Table 42a. Also, the values of Q[j] and QM[j] are resolved in parallel to reduce delay. This is made possible by placing additional drivers between the Q[j] and QM[j] memristors and the intermediate memristors. Lastly, the least significant bit of Q[j] and QM[j] only requires 4 memristors rather than 6 as shown in Table 42b.

The only remaining logic is for the correction of the remainder. However, this correction can be performed in the existing circuitry by altering the multiplexer driver behavior. On the final correction, rather than using both Q_{k-2} and Q_{k-1}, only Q_{k-1} is used to select either the divisor or the value 0. If Q_{k-1} of the remainder is 1, the divisor is selected as operand B, else 0. A final addition now corrects the remainder after 2k+19 steps. Because the design just alters the driver behavior in the final iteration, no additional memristors are required.

In total, the complexity of the design is 17(k+1)-2 memristors and 18(k+1) +4 drivers. The schematic design for this implementation is shown in Figure 116.

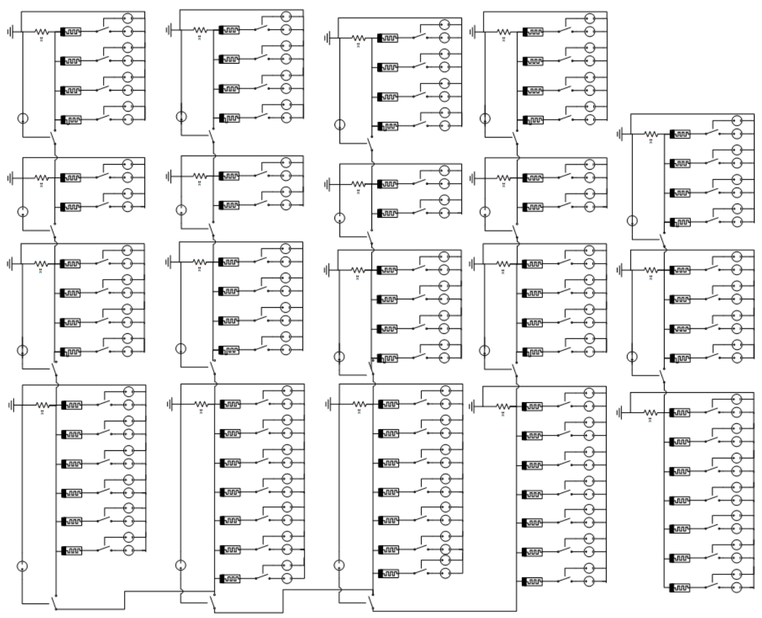


Figure 116: 4-bit IMPLY SRT divider schematic

The top two rows are responsible for the on-the-fly conversion. The third row holds the multiplexer and register logic for the divisor and dividend. The final row is the adder. This design is less than 20% of the baseline design which required 87k-2 memristors and 87k-2 drivers. The overall delay of the final design is now (X+2) (2k+25)-5. Note that although the on-the-fly conversion requires 7 steps (more than the original multiplexer for selecting operand B), this is not on the critical path for completion since it can be hidden by the next addition and the final correction latency. This design reduces the delay of the baseline implementation by about 20x. The complexity and delay comparisons are shown in Table 43.

Table 43: Delay and complexity comparisons for baseline and proposed IMPLY implementations of the SRT divider

	Baseline	**Optimized**
Area	87(k+1)-2 memristors 87(k+1)-2 drivers	17(k+1)-3 memristors 18(k+1)+4 drivers
Delay	2k+58+(X+1)(40k+21)	(X+2)(2k+25)-5

HYBRID-CMOS SRT DIVIDER

The hybrid-CMOS approach has a total complexity of 94k+36 memristors and 48k+28 MOSFETs and a delay of 2k+18 per iteration. The schematic for a 4-bit SRT division circuit is given in Appendix D since its appearance at the gate level is nearly identical to CMOS.

The circuit consists of two 5-bit adders, a 4-to-1 5-bit mux, a 5-bit shift register, and two 5-bit registers. The first adder computes the 2's complement of the divisor. The multiplexer selects whether the divisor or 2's complement of the divisor should be used as the input operand to the next addition. The subsequent adder performs the iterative additions to execute the division. Upon completion of the addition, the result is stored into the shift register and then shifted left. At this time, the upper two bits are used as select lines to the feed the multiplexer and adder.

The on-the-fly conversion is performed using the two 5-bit registers, one to hold Q[j] and one to hold QM[j]. At the end of each iteration, the top two bits of the shift register are used to select either the current Q[j] or QM[j]. They are also used to produce the current least significant bit of each register.

Each 1-bit adder and each 2-to-1 1-bit multiplexer must be buffered at their outputs. Also, each D-latch in the registers must be buffered due to the feedback. In total, the hybrid-CMOS SRT circuit requires (6k+6) (6 memristors + 4 MOSFETs) for the registers, 4 MOSFETs + 30k memristors for the multiplexers, and 28k memristors + 24k MOSFETs for the adders for a total of 94k+36 memristors and 48k+28 MOSFETs.

An example of the first two iterations of the division 0.0101 by 0.1100 is shown in Figure 117.

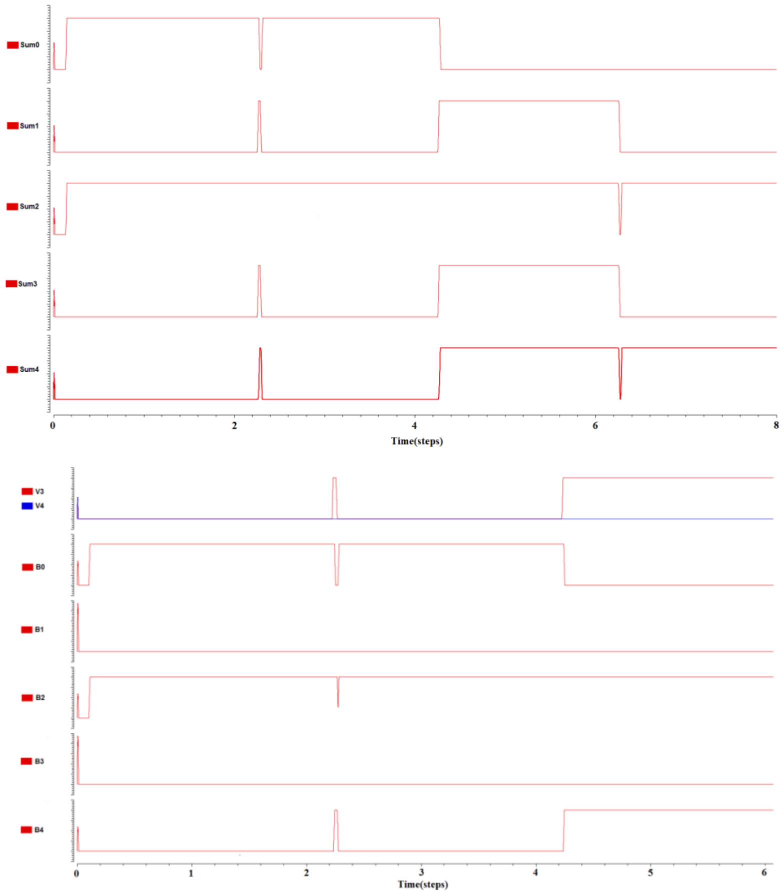


Figure 117: IMPLY SRT divider execution for 0.0101 / 0.1100

In the first iteration, the value of the dividend, 0b0.0101 is effectively loaded into the shift register. Then, the value is shifted. The mux correctly identifies the upper two bits as 0b01 and selects the 2’s complement of the divisor as the operand to the adder. On the next iteration, this results in the sum 1.1110 at the end of the second iteration.

In each iteration of the SRT division, the critical path consists of an addition, storing into the shift register, performing a shift, and multiplexing the result for Q[j], QM[j], and the second operand to the next addition. In total this is 2k+18 gate delays per iteration, so for an X-iteration SRT division, the total delay is (X+2) (2k+18) gate delays.

THRESHOLD GATE SRT DIVIDER

The proposed threshold gate-based design for a SRT divider requires 52k-12 memristors, 15k-3 GOTOs, and 6 MOSFETs. The presented optimized ripple carry adder is used in the SRT for the 2's complement computation. However, the subsequent logic in the SRT design can be optimized to incorporate the multiplexer with minimal additional delay or area. Figure 118 shows the design of a full adder with the proposed combined multiplexer-adder functionality.

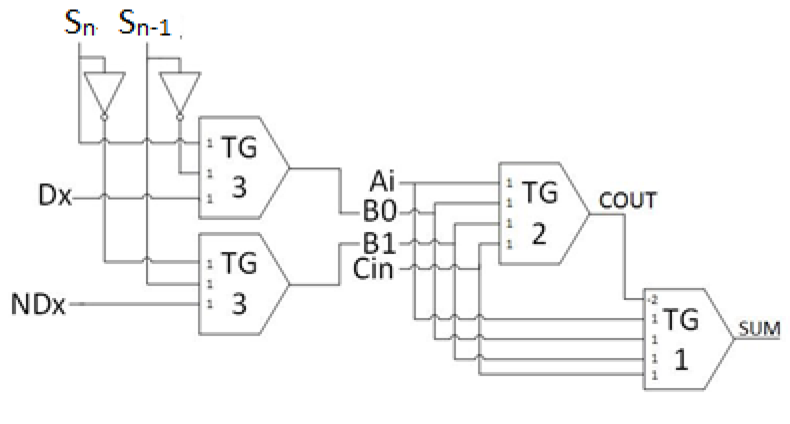


Figure 118: Threshold gate multiplexer-adder functionality for SRT division

Rather than the traditional full adder with inputs A, B, and carry-in, this adder has the inputs A, B0, B1, and carry-in. B0 represents the scenario where the two most significant bits of the sum (S_k and S_{k-1}) are 0b10 and the divisor D is selected. B1 represents the scenario where S_k and S_{k-1} are 0b01 and the 2's complement ND is selected. These events have equal weight and are mutually exclusive, ensuring correctness of the modified full adder. This optimized full adder only adds 2(k+1) memristors to the (k+1)-bit adder and 6(k+1) memristors, 4 MOSFETs, and 2(k+1) GOTO pairs for the multiplexers. Without this optimization, the adder would use k+1 more memristors and have 1 more step. The total delay for the mux and adder is k+4 steps.

This divider also uses the shift register cell with load capability presented for the binary non-restoring divider. In total, these cells use 4 MOSFETs, 5(k+1) GOTO pairs and 16(k+1) memristors and have a delay of 3. In this design only 2(k+1) additional flip flops for the Q[j] and QM[j] values are necessary. The threshold gate circuits shown in Figure 119 correctly evaluate Q[j] and QM[j].

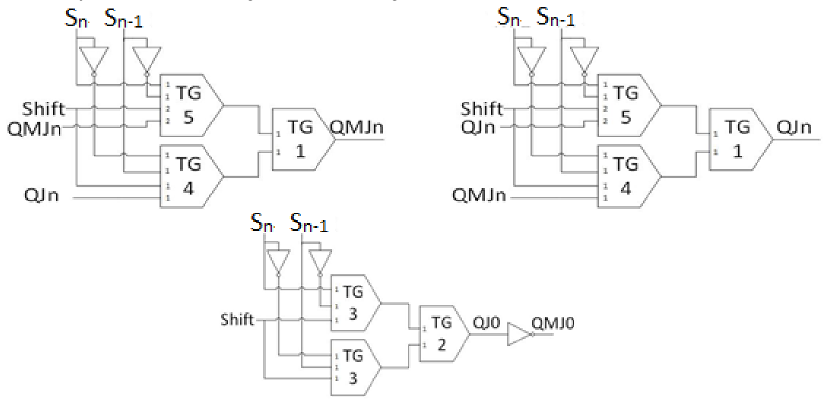


Figure 119: Threshold gate circuits for Q[j] and QM[j] registers

Note that the 0th bit of each register is a special case, simplified given the knowledge that it will only be set in the final iteration and not used as input in a subsequent iteration. When the shift signal is high, this indicates that the most significant bits of the result register can be correctly sensed and used as select lines for the Q[j] and QM[j] registers. Note that for bit 0, rather than evaluating both Q[j] and QM[j] in parallel circuitry, the result of Q[j] is simply NOTed to obtain QM[j]. This is done because the evaluation of these registers is not on the critical path so an additional gate delay can be sacrificed in favor of lesser area.

DeMorgan's law is used to simplify the equation for the registers and reduce the delay from 3 to 2. The delay per iteration for the design is the delay for the multiplexer, adder, load, and shift functionality. For a division which executes X iterations, this is a total delay of (X+2) (k+10). The total complexity of the design is 58k+39 memristors, 17k+14 GOTOs, and 10 MOSFETs.

MAD GATE SRT DIVIDER

The proposed MAD-based design is optimized to require a delay of only k+3 steps per iteration with a complexity of 8(k+1) memristors, 20(k+1) switches, and 9(k+1)+1 drivers. This design uses the adder presented that is capable of performing an n-bit addition and storing the sum into a left shifted position in the adder for a second iteration. This adder requires 4(k+1) memristors, 13(k+1) switches, and 3(k+1) +1 drivers and completes in k+2 steps. The adder is also used for the initial computation of the divisor’s 2’s complement. For the 2’s complement, the MAD drivers are altered to store the result into a separate memristor register rather than into a shifted position so that it can be recalled on later iterations. Similarly, when the inverse of the divisor is first loaded into the A operand of the adder prior to computing the 2’s complement, the divisor is also stored into a separate register for recall. This adds 2(k+1) memristors.

The multiplexer functionality is implemented with MAD gates. Recall that if the result is b00 or b11, then 0 should be loaded into B, if the result is b01 then the 2's complement of the divisor should be loaded and if it is b10, then the divisor should be loaded. Thus, during the multiplexer step, the registers which hold the divisor and its 2's complement are both given V_{cond} signals, gated by their respective bit in the result. This requires two switches, one per register. The two most significant bits of the result are also applied the V_{cond} signal so they can be properly sensed by the mux drivers. If the value of S_k is a logical 1 then the value of the divisor is gated and if the value of S_{k-1} is a logical 1 then the 2's complement of the divisor is gated. However, this could result in both values being mistakenly gated. To overcome this, two parallel gates are added to each memristor. One is gated by NS_k and the other by NS_{k-1}. As long as S_{k-1} or S_{k-1} is 0, the value is stored into the operand B. In all, 3 switches are required on each B input memristor. A schematic showing this driver functionality is shown in Figure 120.

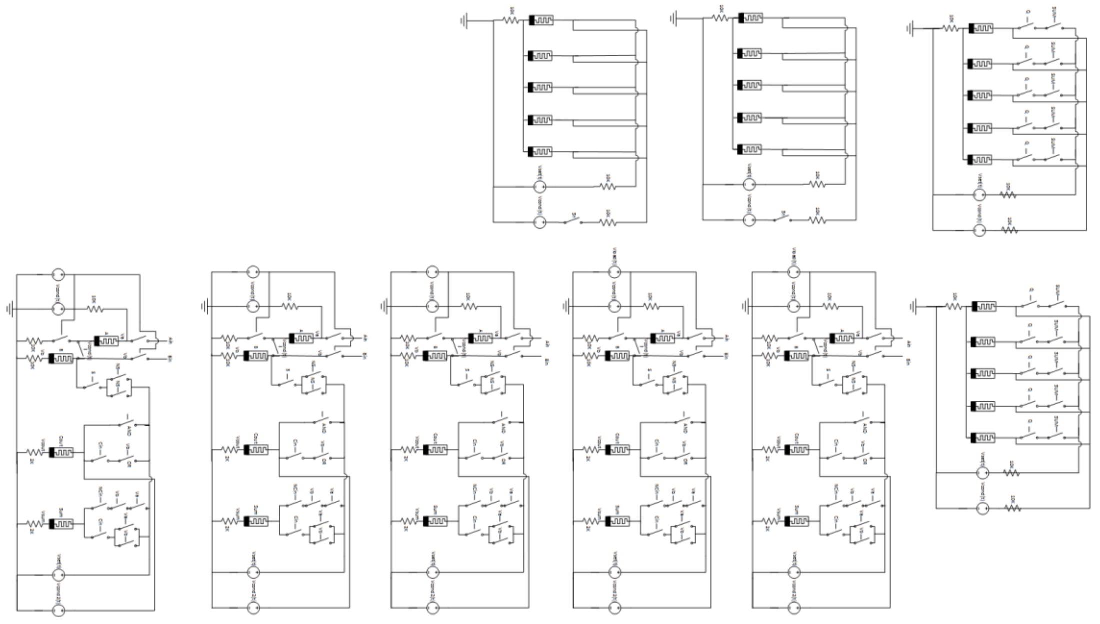


Figure 120: MAD SRT divider schematic

The schematic shows the 5-bit adder in the bottom row, the 2 5-bit registers for the divisor and it's 2's complement in the top row, and 2 5-bit registers for the on-the-fly-conversion in the far-right column. The corresponding drivers for these memristors as described are shown above the memristors. A waveform for an example of the SRT division is shown in Figure 121.

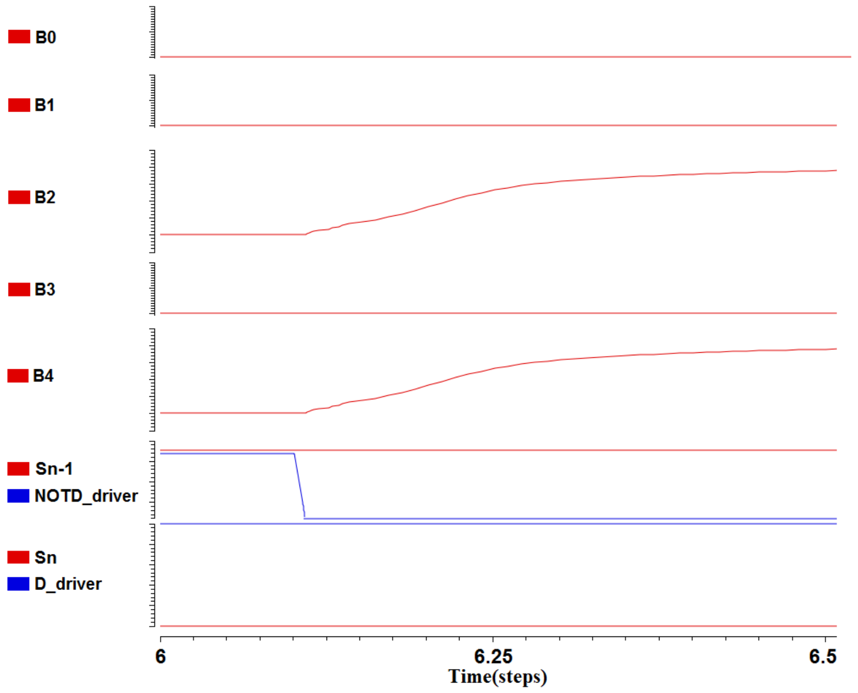


Figure 121: MAD SRT divider mux functionality in the SRT divider for selecting the 2's complement of the divisor in the first iteration of the division 0b0.0101 / 0.1100

Since the value of the dividend is resolved and stored in operand A at step 6, the multiplexer can execute in the next cycle. The value of the most significant bit of operand A, S_k, is 0 and the value of S_{k-1} is 1. This causes the driver for the 2's complement memristors to apply V_{cond} and set the operand B to 0b1.0100 accordingly. Each iteration consists of the multiplexer and an addition for a total of k+3 cycles per iteration for X+1 iterations including the initial 2's complement.

The MAD-based approach to the on-the-fly conversion is able to greatly reduce the overhead in terms of both area and delay. To perform the on-the-fly conversion, the design has k+1 memristors for the Q[j] value and k+1 memristors for the QM[j] value. In parallel with the multiplexer functionality, these values are updated since these values depend on the 2 most significant bits of the value S too. If the most significant bits of the sum are b10, then the drivers of the $Q[j]_{k-1:1}$ memristors are driven a V_{reset} signal followed by a V_{set} signal and the drivers of the $QM[j]_{k-1:1}$ memristors are driven a V_{cond} signal. If their value is b01, then the drivers of the $QM[j]_{k-1:1}$ memristors are driven a V_{reset} signal followed by a V_{set} signal and the drivers of the $Q[j]_{k-1:1}$ memristors are driven a V_{cond} signal. The resolution logic for $Q[j]_0$ and $QM[j]_0$ is simpler, sensing the same bits but driving based on the XOR and XNOR values respectively. Note that at most one of these events can occur so there is no risk of clearing out a memristor that would need to be read by its counterpart. This not only removes memristors intended for holding duplicates of these values prior to clearing the values but allows for parallelizing the resolution of QM[j] and Q[j]. Each memristor requires 2 switches.

The on-the-fly conversion can complete in just 2 steps. These steps are overlapped with the multiplexer and addition functionality so the latency is hidden and does not lie on the critical path. In order to accommodate this, the baseline adder design is slightly modified. The on-the-fly conversion requires two consecutive cycles where it can sense the value S. However, in the current design, this value is only sensed once by the multiplexer and then the next addition begins on all bits of the adder. Rather than having all bits of the adder start the next addition in parallel, the two most significant bits of the adder delay this by one step. Since the resolution of these bits is delayed by carry propagation in the addition anyways, this adaptation does not incur a penalty on the critical path. This allows for the on-the-fly conversion to complete in two consecutive steps as described.

The final remainder check and correction can be completed without any additional circuitry. Since the design is based on drivers, the multiplexer sense circuitry is simply altered to only depend on the most significant bit of the result rather than the 2 most significant bits. Specifically, the V_{cond} signal is only applied to the most significant bit. The remaining behavior is identical to the sense circuitry for the multiplexer as described.

Overall, the complexity of the MAD-based design is 8(k+1) memristors, 20(k+1)+2 switches, and 9(k+1)+1 drivers. For a division that performs X iterations, the total delay is (X+2)(k+3).

SRT Divider Analysis and Comparison

Table 44 shows the delay and complexity for the proposed SRT dividers.

Table 44: Delay and complexity comparisons for the SRT dividers

	CMOS	IMPLY	Hybrid-CMOS	Threshold	MAD
Delay	(X+2)(2k+10)	(X+2)(2k+25)-5	(X+2)(2k+18)	(X+2)(k+10)	(X+2)(k+3)
Area	88k+4 MOSFETs	17k+14 memristors 18k+22 drivers 28k+19 switches	94k+36 memristors 48k+28 MOSFETs	51k+39 memristors 17k+14 GOTOs 10 MOSFETs	8(k+1) memristors 9(k+1)+1 drivers 20(k+1) +2 switches

The proposed MAD-based design has the lowest delay, with a delay of k+3 steps per iteration of the division. The threshold approach also offers a competitive delay with k+10 steps per iteration. Both are half the delay of the traditional CMOS design for increasing k. The hybrid-CMOS and IMPLY methods both have a O(2kn) delay rather than O(k) but keep the delay to roughly equivalent to CMOS.

The MAD-based design also has the fewest components, requiring only 8(k+1) memristors, 20(k+1)+2 switches, and 9(k+1)+1 drivers. This is about half the number of components in the optimized IMPLY design and the CMOS design. The threshold approach requires 51k+39 memristors, but few GOTO pairs and nearly no CMOS. The hybrid-CMOS approach has a significant increase in component count, using 94k+36 memristors and 48k+28 MOSFETs. The significant use of MOSFETs is due to the need for signal restoration on each bit in the shift register and register feedback loops as well as the outputs of each mux and adder component. This indicates that hybrid-CMOS designs may not be well suited for designs which require numerous feedback loops. However, this amount of CMOS is almost half of the traditional CMOS design which requires 88k+4 MOSFETs. The hybrid-CMOS implementation is able to transform the majority of these MOSFETs into memristors.

Goldschmidt Dividers

Memristor-based Goldschmidt dividers are presented for the IMPLY, hybrid-CMOS, threshold, and MAD contexts.

IMPLY GOLDSCHMIDT DIVIDER

The IMPLY approach to an N-bit Goldschmidt division requires an N-bit adder, 2 N-bit Dadda multipliers, and logic for transferring the results from each component to the others. In addition to the initial shift and subtraction for the estimate of $F_{0,}$ each iteration, the 2's complement of the divisor is computed and subtracted from the value 2. Then the two multiplications execute. The initial shift operation requires 2 steps and the adder requires 2N+19 steps and 7n+1 memristors. The optimized 8-bit Dadda multiplier is used which requires 385 memristors, 427 drivers, 482 switches, and a delay of 106 steps. The copy logic from each multiplier unit to the adder and from the adder to the multipliers requires an additional 3n memristors and 4 steps. Thus, the total delay for a baseline optimized 8-bit Goldschmidt divider that uses X iterations is 180X+2 with a componential area of 851 memristors plus driver circuitry. This work reduces this delay to 130X and the componential area to 782 memristors plus driver circuitry.

First, the N-bit adder can be optimized to leverage the context of a Goldschmidt divider. Note that the subtraction and 2's complement always have one known operand. Table 45a and 45b show how the IMPLY full adder can be simplified when the value of an input operand, B, is given.

Table 45a: Steps for an optimized IMPLY parallel add operation, A+B for B_n=1

STEP	FUNCTIONALITY	GOAL
0	Load A	Load A, M2 (A AND B)
1	A imp 0	A->M0 (A XOR B)
2+2i	C_{in} imp ((A XOR B) imp 0)	C_{in}->M2
3+2i	(C_{in} AND (A XOR B)) OR AB-> **C_{OUT}**	M2->A
4+2i	*Next bit reads C_{out} result*	
5+2i	C_{in} imp 0	C_{in} -> M1
6+2i	C OR (A XOR B)	M1->M0
7+2i	*Next bit reads C_{out} result*	
8+2i	(C OR (A XOR B)) imp 0	M0->M3
9+2i	(C NAND (A XOR B)) imp ((C OR (A XOR B)) imp 0)	M2->M3
10+2i	(C NAND (A XOR B)) imp ((C OR (A XOR B)) imp 0) imp 0 = **SUM**	M3->**M4**

Table 45b: Steps for an optimized IMPLY parallel add operation, A+B for B_n=0

STEP	FUNCTIONALITY	GOAL
0	Load A	Load A (A XOR B)
1	A imp 0	A->M2 (A XNOR B)
2+2i	C_{in} imp ((A XOR B) imp 0)	C_{in}->M2
3+2i	(C_{in} AND (A XOR B)) OR AB-> **C_{OUT}**	M2->M0
4+2i	*Next bit reads C_{out} result*	
5+2i	C_{in} imp 0	C_{in} -> M1
6+2i	C OR (A XOR B)	M1->A
7+2i	*Next bit reads C_{out} result*	
8+2i	(C OR (A XOR B)) imp 0	A->M3
9+2i	(C NAND (A XOR B)) imp ((C OR (A XOR B)) imp 0)	M2->M3
10+2i	(C NAND (A XOR B)) imp ((C OR (A XOR B)) imp 0) imp 0 = **SUM**	M3->**M4**

If B is 1 the logic simplifies from A XOR B to NOT A and A AND B to A. If B is 0, the equations simplify from A XOR B to A and from A AND B to 0. These operations can each be done in a single step. Carry propagation can begin at step 2, resulting in a total delay of 2N+8 (as compared to 2N+19) and reducing the complexity to 6n+1 memristors. Since the carry-in is known to be 0, the memristor for this value can be removed, further reducing the area to 6n memristors.

This optimized adder can be used for any addition in which some or all of the bits in an operand are known. This divider can leverage this adder to remove the 2's complement and optimize the operation 2-D in each iteration. Note that the operation 2-D is equivalent to 2 + ((NOT D) + 1). For the purposes of the divider, the adder is set to have two bits for the integer and 6 bits for the decimal. Thus, in order to add 1 to the reciprocal of D and add the integer value 2 to the result, the value h81 is added to the reciprocal of D. Therefore, instead of performing a 2's complement operation followed by an addition, the design can use the proposed adder to perform a single add: h81 + the reciprocal of the divisor. Thus, the IMPLY drivers for bits 0 and 7 of the adder will use the steps from Table 45a and the remaining bits will use the steps from Table 45b. A waveform for the operation 2-D for D=5/8 using the proposed adder is shown in Figure 122.

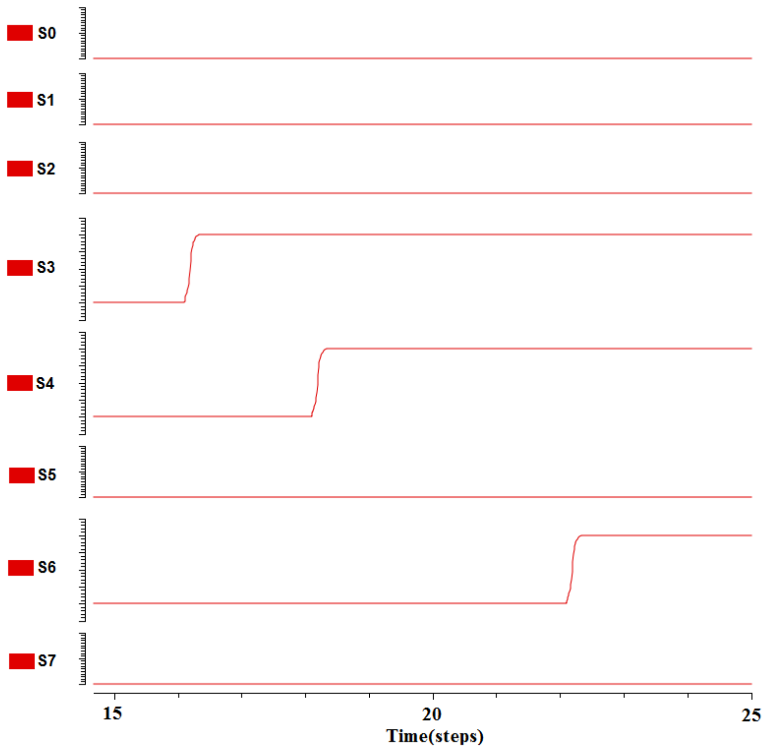


Figure 122: 8-bit IMPLY optimized adder execution for 2-D for D=5/8

The bits resolve every two steps, beginning at step 10 and completing in step 24. The adder correctly resolves the value to 1 3/8, where bits 6 and 7 are the integer portion and bits 0-5 are the fraction.

This adder can be optimized to perform the initial estimate of F too. The estimate, F_0= 3 - 2D, involves a shift and subtraction. F_0= 3 - 2D is equivalent to F_0= 3 + (NOT 2D + 1). Thus, for the initial estimate of F_0, the IMPLY drivers for bits 0, 6, and 7 will use the steps from Table 45a and the remaining bits will use the steps from Table 45b to emulate the operand B=11.000001. Also, the initial value of D will be shifted upon its load into the adder to accomplish the multiplication by 2. This optimization removes the need for a second adder and shift module.

As a third optimization, the result of the addition can be stored directly into the multiplier inputs to remove the need for the sum memristors in each bit and remove two copy steps from the overall delay. Similarly, the result of the dividend multiplier can be stored back as the first operand in the multiplier for the next iteration and the result of the divisor multiplier can be stored directly into the A operand of the adder. This saves another 2 steps and 28 memristors.

Coupling the optimized adder with the optimized Dadda multipliers and establishing the correct connections between the units produces the final schematic in Figure 123.

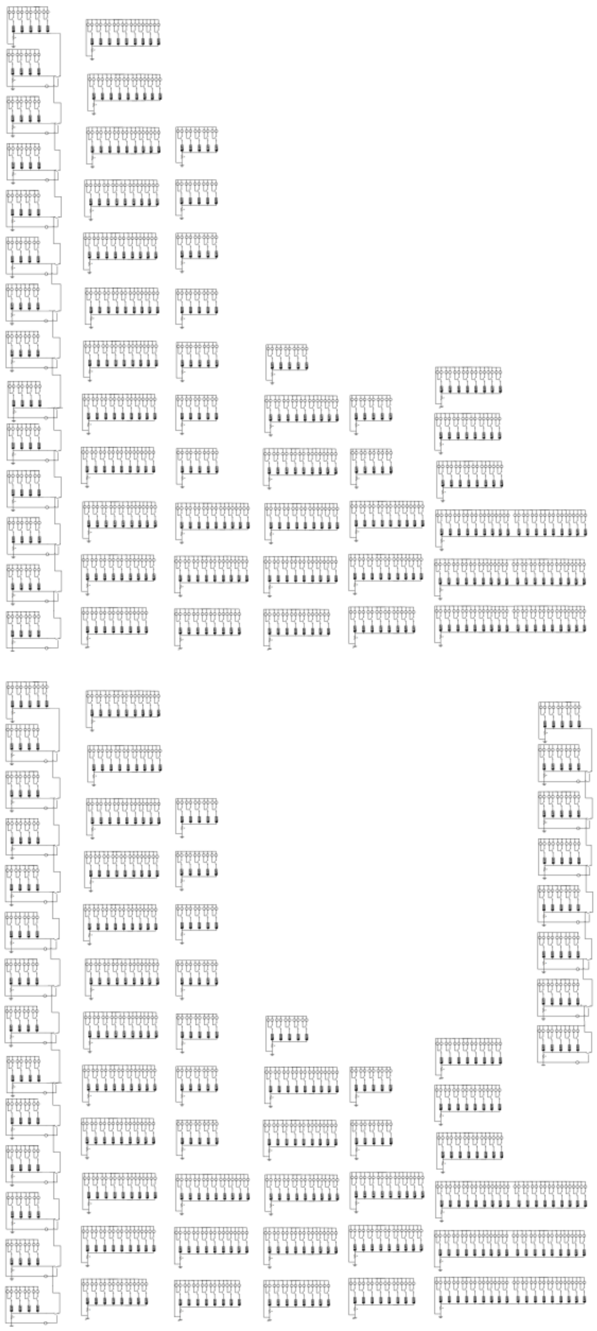


Figure 123: IMPLY Goldschmidt divider

Note that it is not possible to overlap the area required for the two Dadda multipliers even though both have a common input parameter, F_i. This is because the first step of the multipliers is to perform a matrix of AND operations. This creates disjoint inputs to the remaining stages of full adders and half adders, preventing any operation overlap. However, it is possible to share the drivers across the two instantiations since they perform their operations in lockstep. This removes 427 drivers from the design. Also, it is not possible to pipeline divisions since the circuitry is already pipelined for the individual iterations.

HYBRID-CMOS GOLDSCHMIDT DIVIDER

The hybrid-CMOS approach to a Goldschmidt divider is designed to minimize the use of CMOS while optimally reducing delay. The proposed design requires 2000 memristors and 1444 MOSFETs and a delay of 104 steps for per iteration of the division.

The design consists of an 8-bit adder and 2 8-bit Dadda multipliers and latches to store values between the units. However, the adder cannot be optimized in the way it was for the IMPLY Goldschmidt divider to leverage knowledge of the inputs. This is because in the IMPLY context, this results in a simple memristor driver change without a change to the circuitry. In the hybrid-CMOS context, the actual circuitry would be changed. Since the initial estimate and the remaining additions do not use the same known operands, there is no circuitry that would accommodate both operations. Thus, the standard ripple carry adder which requires 14N memristors + 12N MOSFETs with a delay of 3N+4 is used. An alternative would be to use two adders, one for the initial estimate and one for the remaining additions. Each could have optimized circuitry for its known inputs to reduce delay but this adds area to the design. This work optimizes for area over delay in this case since hybrid-CMOS tends to suffer from high area.

In the context of the Goldschmidt divider, the sum output of the 8-bit adder serves as input to both Dadda multipliers. Within these multipliers, each bit of the sum serves as an input to n gates in the initial NxN AND gate array in each multiplier. Thus, each bit of the sum output has a fan-out of 2N. Since the fan-out is greater than 3, each sum output in the adder must be buffered and these MOSFETs cannot be removed from the adder.

Figure 124 shows the waveform for the operation 2-D for D=5/8 for the adder with buffering. Again, this operation is achieved by adding the reciprocal of D to h81.

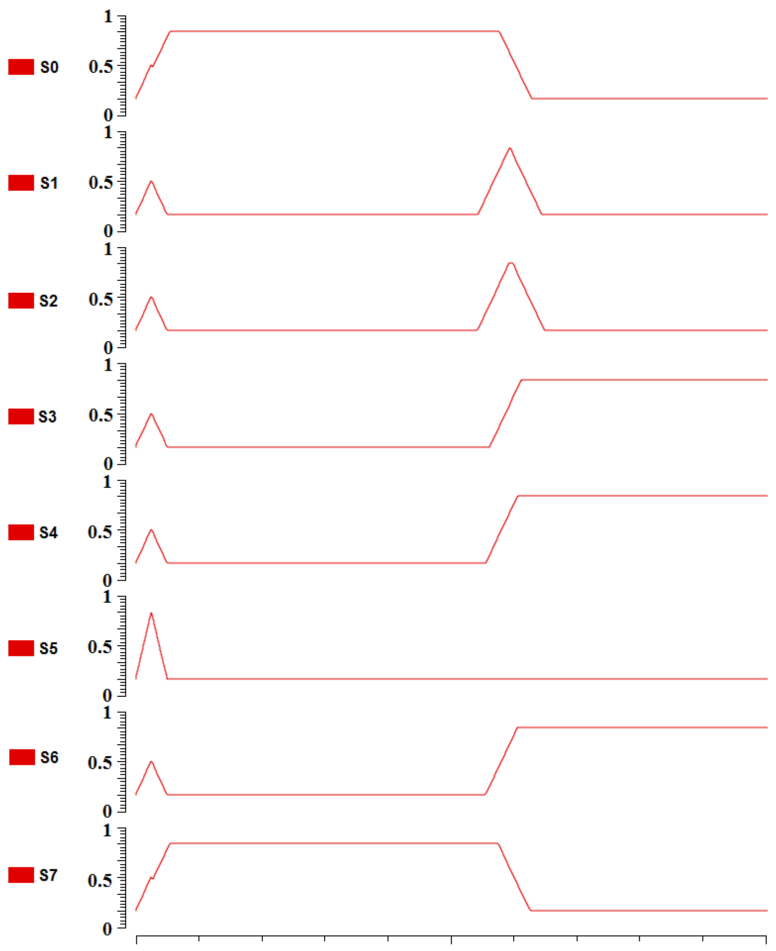


Figure 124: Hybrid-CMOS adder execution for 2-D for D=5/8

As shown in previous work, each of the 8-bit Dadda multipliers requires 856 memristors and 602 MOSFETs with a delay of 72. The output of the multipliers, D_i and N_i, must be latched at the end of each iteration. On the next cycle, the reciprocal of D_i can feed the adder and D_i and N_i can feed as inputs to their respective Dadda multipliers. The flip-flops provide the reciprocal of D_i in parallel with D_i without an extra delay. Each flip-flop requires 12 memristors and 8 MOSFETs with a delay of 4. A complete schematic for the optimized 8-bit hybrid-CMOS Goldschmidt divider is shown in Figure 125.

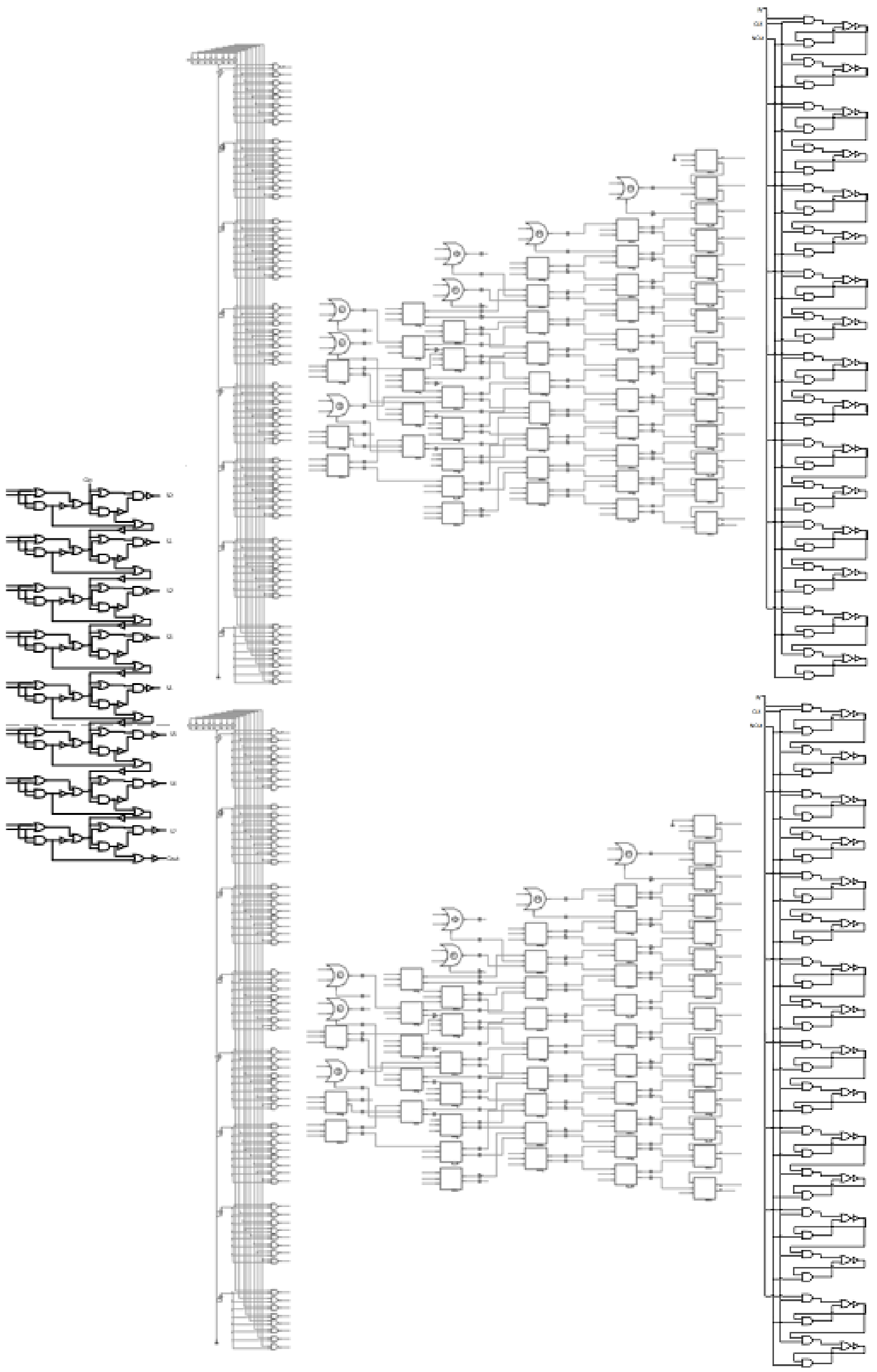


Figure 125: 8-bit Hybrid-CMOS Goldschmidt divider schematic

The total complexity for the hybrid-CMOS approach is 2000 memristors and 1444 MOSFETs. The delay is 28 steps for the adder, 72 for the multipliers, and 4 for the flip-flop, for a total delay of 104 steps for each iteration of the division.

THRESHOLD GATE GOLDSCHMIDT DIVIDER

The threshold gate Goldschmidt divider performs similar optimizations as the other designs to achieve lower latency and area. First, the N-bit adder can be optimized for known inputs. Figure 126 shows the implementations for the optimized threshold-based full adders for the case when input b=0 and b=1.

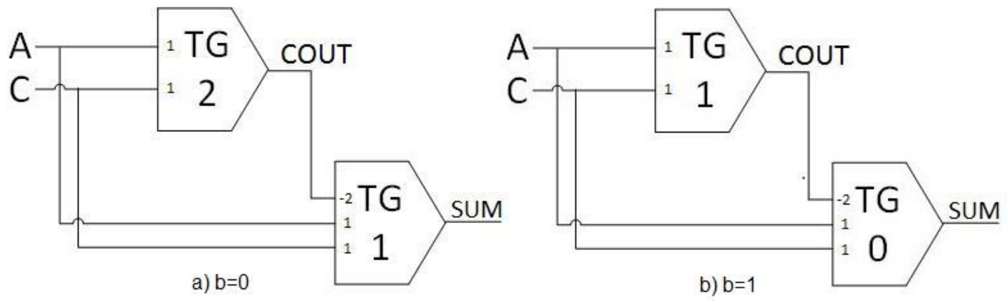


Figure 126: Optimized threshold full adder for a) b=0 and b) b=1

The complexity of each full adder is reduced from 7 memristors and 2 GOTO pairs to 5 memristors and 2 GOTO pairs. The delay remains unchanged. An N-bit adder constructed from these gates requires 5N memristors, 2N GOTO pairs, and has a delay of N+1 gate delays. However, since the circuitry itself changes, the design would require two adders for a total of 10N memristors and 4N GOTO pairs as compared to the single baseline adder with a complexity of 7N memristors and 2N GOTO pairs. Since the two options have equivalent delay, the single baseline adder is selected. For an 8-bit adder, the componential area is 56 memristors and 16 GOTO pairs with a delay of 9 gate delays.

Prior work showed that a threshold gate based 8-bit Dadda multiplier requires 379 memristors and 174 GOTO pairs with a delay of 11 steps. A flip flop can be implemented with 4 GOTO pairs and 10 memristors. Two N-bit flip flops are needed, one to latch the result of each multiplier. Thus, for an 8-bit Goldschmidt divider, the total complexity is 1074 memristors + 428 GOTO pairs and the delay is 23 steps per iteration.

MAD GATE GOLDSCHMIDT DIVIDER

The 8-bit Goldschmidt divider based on MAD gates requires 583 memristors, 74 drivers, and 1426 switches and a total delay is 28 steps. As in the previous designs, the divider consists of an N-bit adder and two N-bit Dadda multipliers. Flip flops are not necessary for the design because the values are retained in the memristors themselves between iterations.

The N-bit adder can be optimized given that the input B is known. Each bit of the adder only requires 2 memristors. The most significant bit only requires one memristor since the carry-out signal is not needed. The MAD-based adder is shown in Figure 127.

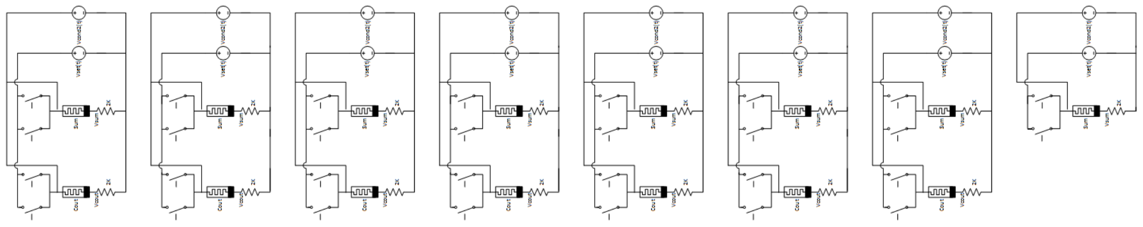


Figure 127: 8-bit MAD optimized adder schematic for known input B

Again, the equations for the carry-out and sum when the B input is 0 are $C_{out} = AC_{in}$ and $S = A \text{ XOR } C_{in}$. Using the MAD-based driver approach, the C_{out} memristor is resolved by gating the A input with the C_{in} value. Thus, the memristor will only be set if the C_{in} value is high and the A input is high. The sum memristor has two parallel inputs:

the A input gated by NOT C_{in}, and the NOT A input gated by the C_{in} signal. The equations for the carry-out and sum when the B input is 1 are C = A or C_{in} and S = A XNOR C_{in}. The carry-out signal is resolved by gating the V_{in} signal with two parallel switches: one gated by the C_{in} value, and the other by the A input. The sum memristor has two parallel inputs: the A input gated by C_{in}, and the NOT A input gated by the NOT C_{in} signal. To accommodate both scenarios, the full adder requires 2 memristors, 4 switches, and 1 driver (for the V_{set} and V_{cond} signal). In total, the proposed 8-bit adder requires 15 memristors, 32 switches, and 8 drivers and has a total delay of n steps. An example of the functionality of this adder for the operation 2-D for D=5/8 is shown in Figure 128.

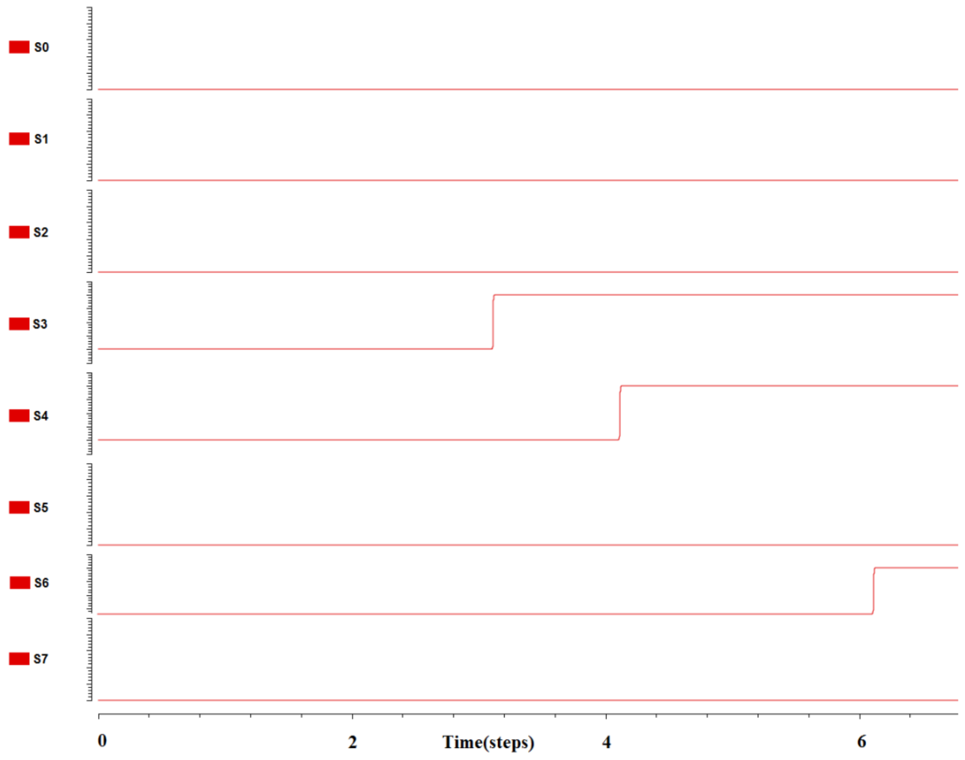


Figure 128: MAD optimized adder execution for 2-D for D=5/8

The sum correctly resolves after 8 steps to 0x58. The 8-bit MAD-based Dadda multiplier requires 298 memristors, 66 drivers, and 697 switches and a delay of 20 steps. The two instantiations of this multiplier can share drivers since they operate in lockstep. This removes 66 drivers from the design. Also, the memristors used to store the result of the multiplications can be removed. Instead, the sum can store directly into the inputs in the adder. In all, this removes 28 memristors from the design. The divider is complete with the addition of driver logic to propagate values from the adder to the multiplier and vice versa. Because of the driver-based nature of MAD designs, this requires no additional area. The final divider is shown in Figure 129.

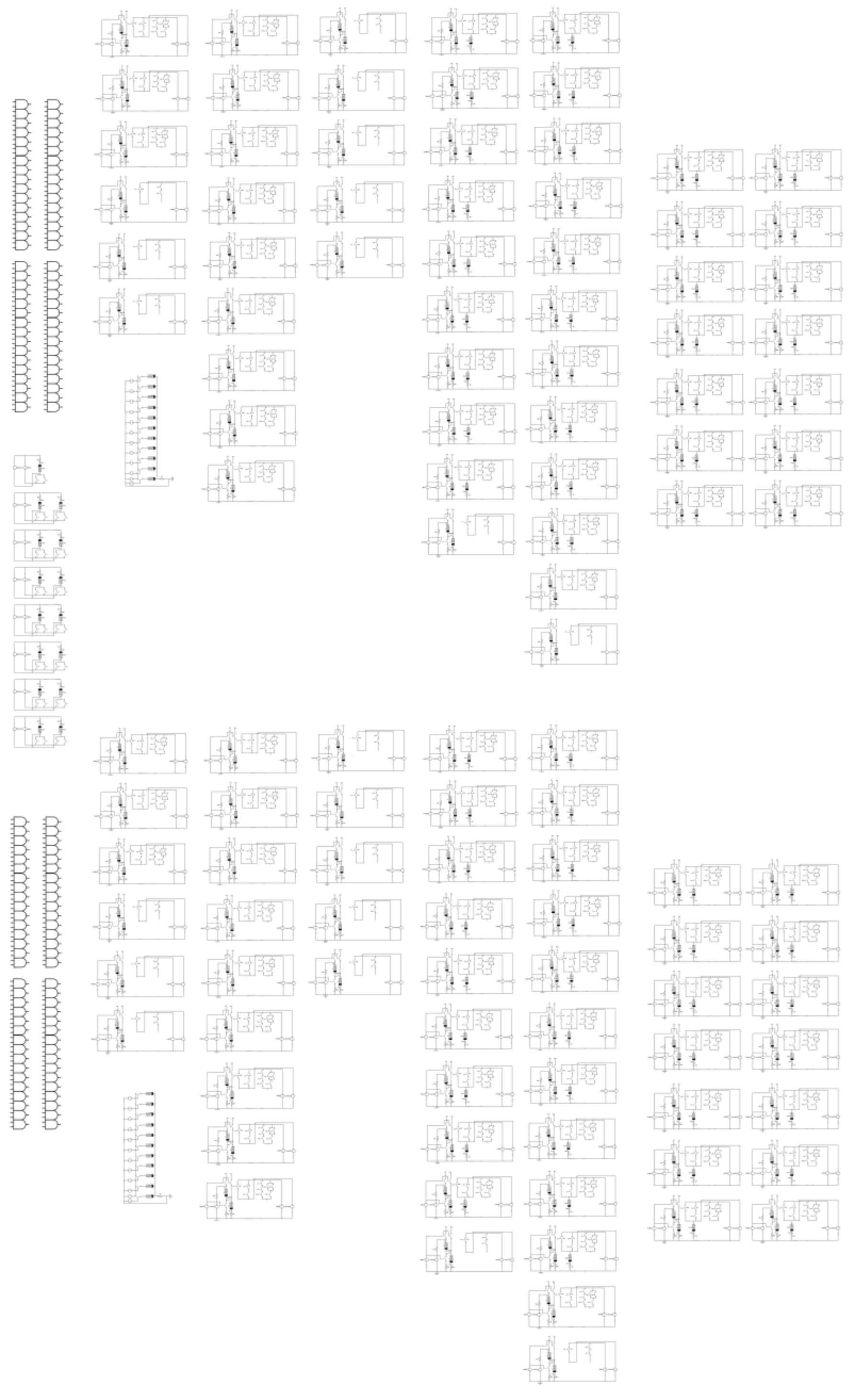


Figure 129: 8-bit MAD Goldschmidt divider schematic

The total complexity of the design is 583 memristors, 74 drivers, and 1426 switches. The total delay is 28 steps. Similar to the binary non-restoring divider, it is not possible to pipeline divisions since the adder and multiplier are pipelined across iterations of a single division.

GOLDSCHMIDT DIVIDER ANALYSIS AND COMPARISON

The complexity and delay breakdown for the proposed Goldschmidt dividers is shown in Table 46.

Table 46: Complexity and delay comparison for the proposed 8-bit Goldschmidt dividers

	CMOS	IMPLY	Hybrid-CMOS	Threshold	MAD
Delay	79X	130X	104X	23X	28X
Area	4968 MOSFETs	782 memristors 451 drivers 988 switches	2000 memristors 1444 MOSFETs	1074 memristors 428 GOTO pairs	583 memristors 74 drivers 1426 switches

The threshold gate and MAD gate based designs are significantly faster than the other proposed designs. Each iteration of the division only requires 23 steps in the threshold design and 26 in the MAD design. Both the CMOS and the hybrid-CMOS approaches have a total delay over 3x that. The additional delay for the hybrid-CMOS over the CMOS design is due to the Dadda multiplier buffering. The IMPLY-based design is the slowest of the proposed designs, requiring 130 steps each iteration.

In terms of area, the IMPLY and MAD approaches have the fewest components, requiring only 782 memristors and 583 memristors and their accompanying drivers. Although the Dadda multipliers introduce a significant number of switches to both designs, the number of drivers is kept relatively low by sharing them. The threshold gate based Goldschmidt divider requires more memristors but fewer overall components.. The hybrid-CMOS approach has the greatest componential area with 2000 memristors and 1444 MOSFETs. It is also the only design which requires any CMOS. However, the amount of CMOS is reduced to less than 1/3 of the traditional CMOS design and the overall componential count is also less than the CMOS design.

CROSSBAR LOGIC[6]

This section presents two alternative memristor implementations to logic-in-memory for the crossbar context. First, MAD gates are translated slightly to function properly in a crossbar structure. Second, another logic gate is presented that takes a similar form to the IMPLY circuit but requires fewer steps and does not overwrite inputs. Both are analyzed and shown to have lower delay and higher flexibility than prior memristor logic.

[6] L. Guckert and E. E. Swartzlander Jr., "Modern System Design using Memristors," Chapter in *Advances in Memristors, Memristive Devices and Systems,* Springer-Verlag, Oct. 2016. This section is an extension of previously published work for which Lauren Guckert is the principal author and responsible for all content.

MAD Gates in a Crossbar

This section presents the application of MAD gates in a crossbar context to enable logic-in-memory applications.

IMPLEMENTATION AND ANALYSIS

MAD gates can be transposed into a crossbar structure with a few minor adaptations to the circuitry. First, in the standard MAD Boolean gate, the input memristors are connected in series by connecting their opposite-polarity terminals, i.e. one input memristor's p-terminal is connected to the second input memristor's n-terminal. This is required in order to be able to maintain the values of the input memristors when V_{cond} is applied. Because of memristor fundamentals, if their common-polarity terminals are connected in series, the values of the input memristors can change when V_{cond} is applied. However, in a crossbar structure, the p-terminals of both input memristors necessarily share a common row line, connecting their common polarity. Because of this, the MAD circuitry must be slightly changed as shown in Figure 130.

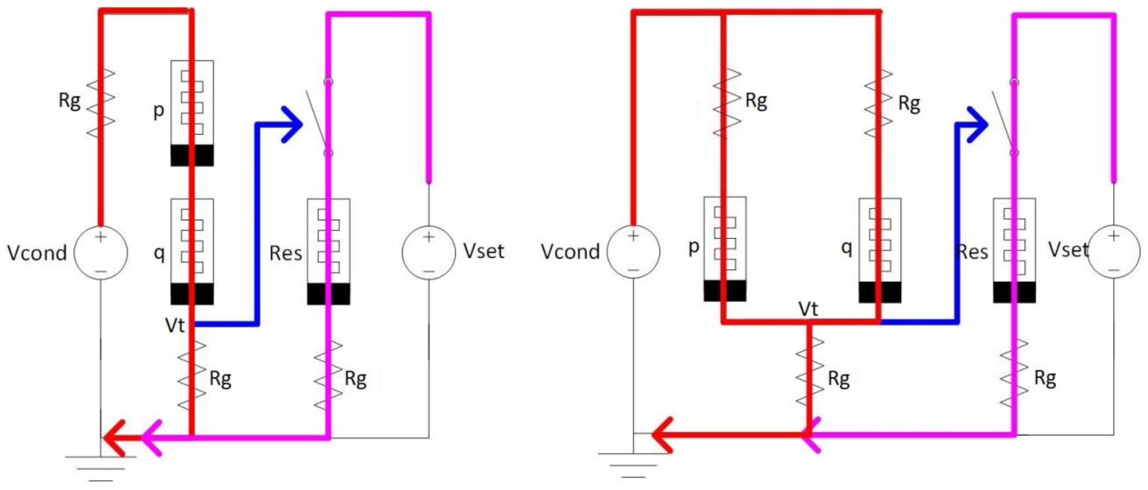


Figure 130: Translation of a) a MAD AND gate into b) a crossbar form

The red highlighted path follows the V_{cond} signal, the purple highlighted path follows the V_{set} signal, and the blue highlighted path follows the voltage division threshold signal. The only necessary change is on the input memristors and the application of V_{cond}. The other signals remain unchanged. The input memristors are still used in conjunction with the pull down resistors to function as a voltage divider circuit, however it takes a different form. Instead of connecting the input memristors in series and applying V_{cond} and GND to the terminals, V_{cond} is applied to the n-terminal of both input memristors and their shared row line is connected to GND via the pull down memristor. The sensed node V_t is now shared by both input memristors. The resistor R_g values are selected as described previously.

Now that the MAD structure has been transformed, it can be inserted into a crossbar. Figure 131 shows the execution of a MAD AND gate in a crossbar structure using the new circuitry.

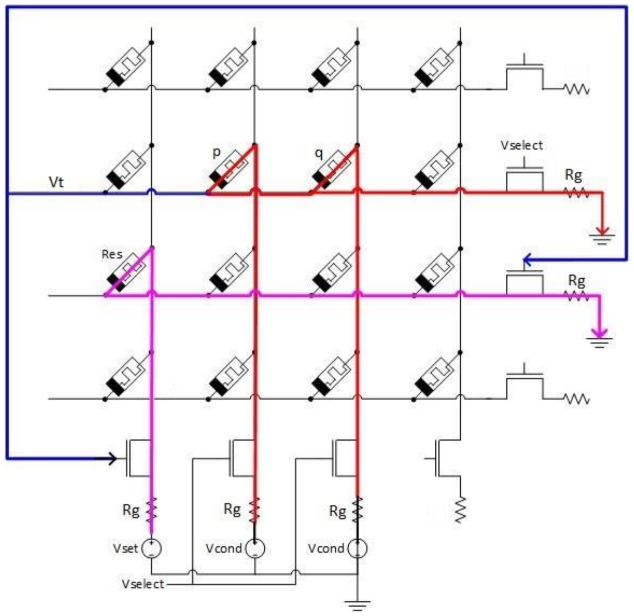


Figure 131: MAD AND gate in a crossbar structure

The inputs p and q are two memristors on the same row in the crossbar. To sense the values of p and q, the row and bit lines of p and q are selected and V_{cond} is driven on both bit lines while the row is grounded. The row line now represents the sensed node voltage, V_t. In the second step, the sensed value at V_t is used to drive the select lines on the result memristor's row and bit lines. If the value of V_t is greater than the threshold of the select line gates, then V_{set} and GND will be driven to the result memristor Res and it will be set to 1. Otherwise, the row and bit lines will not be selected and Res will remain at 0. The switch's threshold voltage is calculated using the same methodology as the standard MAD gates.

For the XOR and XNOR operation, the same crossbar circuit is used but now V_a and V_b are sensed rather than V_t. Figure 132 shows this for the XNOR operation.

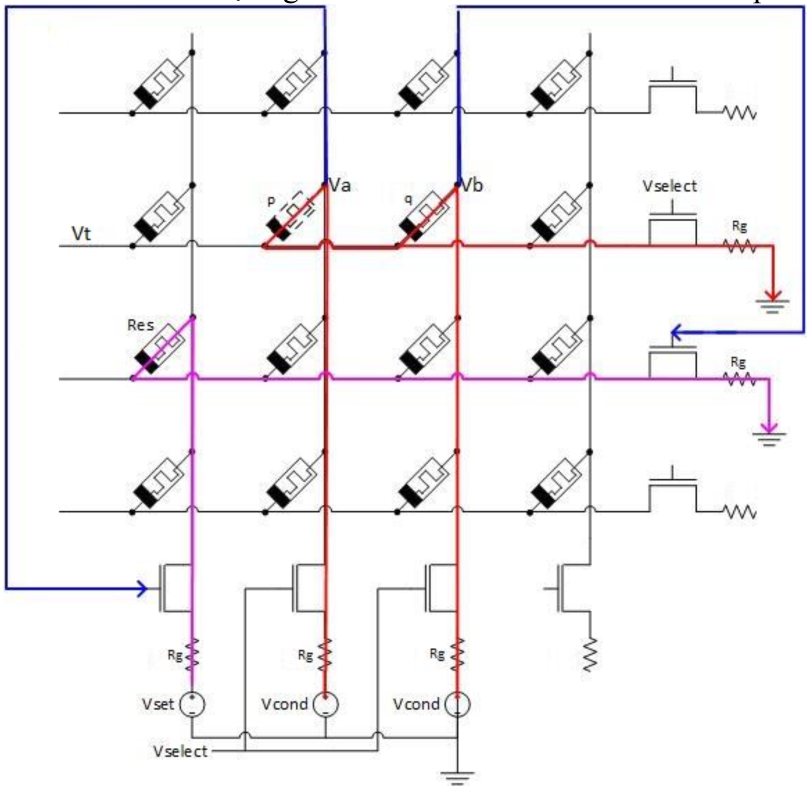


Fig. 132: MAD XNOR gate in a crossbar structure

For example purposes, Let R_g = 2K ohms, low resistance=1K ohms, and high resistance=100K ohms. When both input memristors are 0, the voltages at V_a and V_b are 0.975 V. When input memristor p is a 1 and q is a 0, the voltage at V_a is 0.6 V and the voltage at V_b is 0.982 V. If the values of the inputs are swapped the voltages at V_a and V_b swap accordingly. If both inputs are a 1, the voltages at V_a and V_b are 5/7 V. Thus, V_a and V_b can be used to differentiate the four input scenarios.

In the original XNOR MAD gate, two separate conditions on two switches must both be true to gate the V_{set} signal to the result memristor. This was implemented by placing two switches in series between the V_{set} signal and the result memristor, one gated by V_a (with threshold V_{applyA}) and one gated by V_b (with threshold V_{applyB}). To correctly perform an XNOR, $V_{applyA} = V_{applyB} = 0.6$ V. The result memristor will only be set to 1 when both of these conditions are true, which only occurs when both inputs are 0 or both inputs are 1.

However, this structure is not conducive to the context of a crossbar since it requires two switches. Instead, a crossbar can achieve the same functionality by placing one switch on the bit line gating V_{set} and one switch on the row line gating GND for the result memristor. One switch will still be gated by the voltage V_a and the other by the voltage V_b. This effectively achieves the same operation as the original XNOR gate.

Recall that the NOT and COPY operations only require a single input memristor in the MAD gate design. Since these two gates do not use input memristors in series, there are no issues with the way the terminals of the memristors are connected in the original MAD gate design. Thus, their gate designs do not need to be altered at all to translate into the crossbar context. This can be seen in Figure 133.

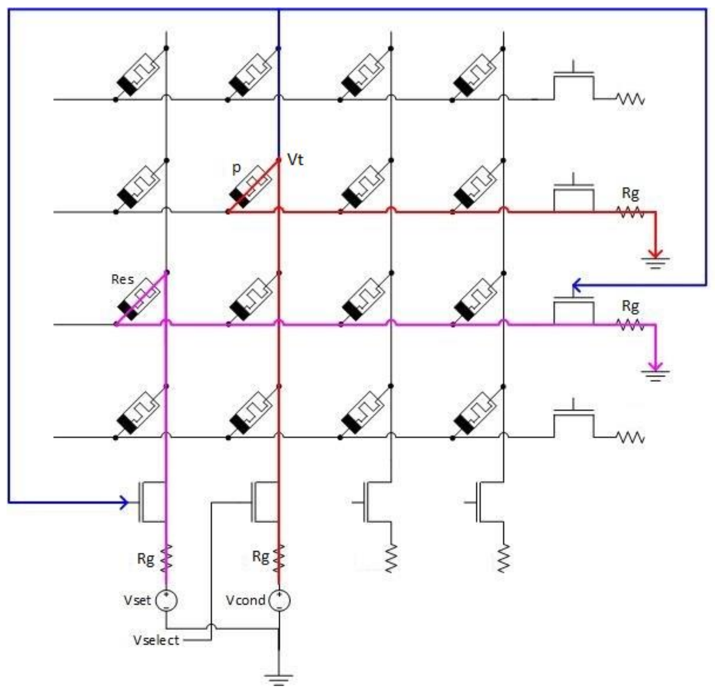


Figure 133: MAD NOT gate in a crossbar structure

V_{select} is driven high for the row and bit line of the input memristor to create a circuit from V_{cond}, through the input memristor and pull down resistor, to GND. In the next step, the voltage sensed at V_t is used to determine if V_{set} should be gated to the result memristor Res. This is identical to the original MAD NOT gate execution.

The IMPLY operation is the most commonly used approach to logic-in-memory. However, the IMPLY operation suffers from long-latency, serialized operations and overwrites inputs during operation. Many other approaches have been proposed to overcome these limitations. Other approaches include MAGIC gates, CRS cells, and more, but none are as established as the IMPLY operation. None of these proposals for memristor crossbar logic, including IMPLY, can perform multiple operations in parallel and each has their own tradeoffs and shortcomings. MAD gates overcome all the issues associated with the other proposals and offer a complete set of logic-in-memory operations with lower area and delay. They also offer a logically complete set of operations that handles high fan-out, does not overwrite operands, and can perform the COPY operation.

MAD gates require only 1 memristor per operand and 2 steps for every Boolean operation. It takes multiple IMPLY steps to achieve most Boolean operations. For example, the XOR operation requires as many as 6 steps and 5 memristors in a crossbar. MAGIC gates have the same complexity and latency as MAD gates, but only the NOR gate has been successfully mapped into a crossbar. Since every other Boolean operation must be performed via a series of NOR operations, the delays are much higher than their MAD counterparts. Zhang, *et al.* propose a novel OR gate that can exist in a crossbar and show how to pair it with another AND gate design and the IMPLY NOT operation to create a logically complete set for logic-in-memory operations. However, these gates destroy one of the operands during operation. MAD gates use an additional memristor to store the result, leaving the inputs intact for subsequent use. Zhang, *et al.* also takes more steps for the NOR and NAND operations and does not offer implementations for the XOR or XNOR operations.

CRS cells have been proposed as an alternative to the commonly used memristor cells in a crossbar to achieve lower delay and area. However, each CRS cell uses two memristors rather than one and requires initialization on all operands before Boolean computation. This incurs an extra delay for each operation. This design also overwrites one of the input operands to store the result. CRS cells are only capable of performing NAND-AND and NOR-OR operations, each of which requires 3 steps as compared to 2 in MAD. All other Boolean operations must be constructed from a series of these operations, requiring a total of N+2 steps where N is the number of NAND and NOR operations required for the Boolean operation.

A full breakdown of the comparison between the MAD crossbar operations with alternative logic-in-memory memristor gates is given in Tables 47 and 48.

Table 47: Latency comparisons for the MAD crossbar and prior crossbar approaches

Operation	**IMPLY**	**MAGIC**	**CRS**	**Zhang et al.**	**MAD**
p NAND q	2	N/A	3	3	2
p AND q	3	N/A	6	2	2
p NOR q	5	1	3	3	2
p OR q	4	N/A	6	2	2
p XOR q	8	N/A	6	N/A	2
NOT p	1	N/A	3	2	2

Table 48: Component counts for the MAD crossbar and prior crossbar approaches

Operation	**IMPLY**	**MAGIC**	**CRS**	**Zhang et al.**	**MAD**
p NAND q	3	N/A	6	2	3
p AND q	4	N/A	8	2	3
p NOR q	6	3	6	2	3
p OR q	6	N/A	8	2	3
p XOR q	7	N/A	8	N/A	3
NOT p	2	N/A	6	2	2

One consequence of the MAD approach is the use of multiple voltages. The V_{cond}, V_{set}, and V_{reset} are all necessary for execution. Although this is equivalent to the number of voltage sources used by the IMPLY operation, other proposed approaches such as MAGIC gates only require a single voltage source. However, this is likely because this approach is only capable of implementing a single operation in a crossbar. If additional Boolean operations were tackled, especially the XOR operation, it is likely that the number of voltages required would increase to match or exceed those required by MAD. The CRS and Zhang, *et al.* approaches both require 4 voltage sources, more than MAD and IMPLY. In Zhang, *et al.*, the first three voltages are similar to those in the IMPLY and MAD approaches, and the fourth voltage has the same magnitude as V_{cond} but opposite polarity.

In addition to low latency and low area, MAD gates also offer increased flexibility over previous approaches. In the IMPLY circuitry, the result memristor inherently must lie on the same row as the input memristors since one of the inputs is used for the output. In a MAD operation in a crossbar, the result can lie on any row line and bit line. This is important for full system designs if logic-in-memory is going to be harnessed for performance gains. If data is frequently being moved around in the memory in order to read, write, and execute logic, the advantages of performing the logic in the memory rather than the CPU is diminished or even eliminated. Operations in the other proposed approaches suffer from this risk, but MAD operations in memory do not.

Also, because the voltages associated with the inputs are sensed in a single step and used as drivers in a latter step, these voltages can be used multiple times for operations which use the same inputs. This can result in increased latency and energy savings by removing the sense step on the input memristors for all operations using the same inputs. For example, if it is desired to execute A AND B followed by A OR B, the first step will send the voltage division circuitry on the input memristors A and B, the second step will drive the write operation for the A AND B result memristor, and the third step will drive the write operation for the A OR B result memristor. In a naïve approach, the sense operation would occur again before writing the A OR B result memristor. Using this knowledge, logic-in-memory operations that use the same inputs should be prioritized to execute consecutively.

Another benefit of MAD gates, and perhaps the most critical, is the fact that the input memristors are not overwritten at any time during execution. This allows for reuse of the operands for later operations if needed. It also retains this data in memory to be read or written later by the system. Since all of the other proposed approaches to crossbar logic overwrite one or more of the input operands during execution, the original data is lost. Thus, if the system were to need this data for another operation or a simple write, it would need to execute additional instructions to reproduce this value. This would involve reading the data out of memory and storing it in auxiliary space, then performing logic-in-memory operations, moving the result of the operations to another location, and finally writing back the original value from auxiliary space to the memory.

Lastly, MAD gates do not require any memristors or steps for holding intermediate values in the calculation of a Boolean operation. All memristors in a MAD gate, whether in a logic or memory context, are either inputs or outputs of the Boolean operation. In other approaches, such as CRS and IMPLY, numerous memristors are required during intermediate steps. This leads to extra memory cells that do not hold any "real" data. This has long-reaching effects that degrade performance, area, and power in full system designs. MAD gates overcome all of these deficiencies.

BOOLEAN OPERATION AND THRESHOLD GATE SELECTION

Depending on the Boolean operation being performed on a MAD gate or MAD crossbar, the threshold voltage, or V_{apply}, of the gated drivers will vary. However, the hardware cannot be easily reconfigured dynamically to alter this value. This is not an issue for traditional logic since their circuitry is determined by their functionality, but the crossbar must adapt to this complexity.

For this reason, a single switch cannot be used to drive a given line in the crossbar. Instead, every row and bit line in the crossbar must have the ability to be driven by multiple switches, each corresponding to a different Boolean operation. The encoding for the Boolean operation will be used to select which one of the input lines is used to drive the given row and column of the result memristor. A visualization of this can be seen in Figure 134.

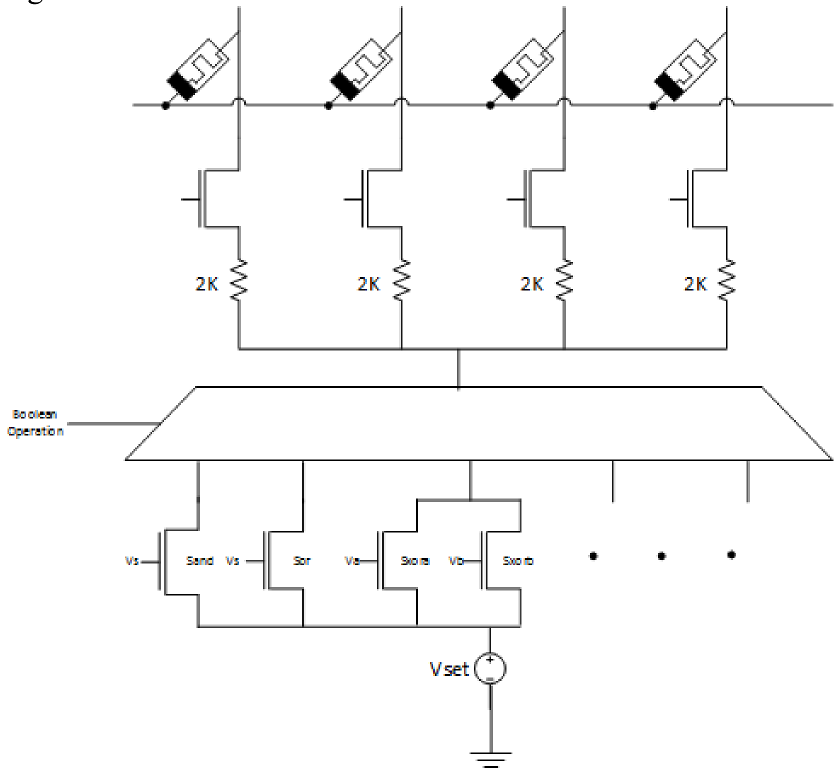


Figure 134: Driver logic for selecting MAD crossbar Boolean operations

Now, the V_{set} signal has multiple paths to the crossbar, one for each Boolean operation. Let the first path correspond to the AND operation. Thus, a single switch, S_{and} will lie on this path and have a corresponding V_{apply}. Let the second path correspond to the OR operation. A single switch, S_{or}, will lie on this path with its calculated V_{apply} value. Both will be driven by the sensed voltage V_s. The third path will have two parallel switches for the XOR operation, S_{xora} and S_{xorb}, driven by the voltages V_a and V_b. The process will continue for each of the remaining operations. The path corresponding to the encoded Boolean operation will be selected and gated to the relevant bit line in the crossbar. Similar logic exists for the row lines.

This adaptation is similar to the mux capabilities that already exist in crossbars for selecting row and column lines when addresses into the memory memristor cells. Thus, this additional mux stage does not contribute to the critical path. Also, this hardware only needs to exist once for the entire memory, contributing negligible area. Also, the other logic-in-memory approaches which require multiple voltage drivers require similar mux behavior. No previous work has implemented or calculated this logic.

The final concern of this design is the sensitivity of the circuitry to correctly differentiate the voltages at the sense nodes to determine the threshold satisfaction on the row and bit lines. For example, for the XOR operation, the switches must be sensitive enough to differentiate 0.975 V vs. 0.982 V in the example.

A simple solution is to reselect the parameterized values of the system, namely the values of low-resistance, high-resistance, and the pull down resistor R_g. For example, by changing the value of the resistors on the row and bit lines, the value of V_{cond}, and the low and high resistances of the memristors, the threshold voltages can be changed. The values selected in this paper were done for simplicity and serve as motivation and do not necessarily represent ideal values. Note that care must be taken when these parameters are chosen to abide by the memristor design methodologies as specified in prior work [19].

Alternative Logic Gates in a Crossbar

This section proposes a novel implementation of a memristor-based logic gate that is designed for seamless integration into a crossbar to allow for logic-in-memory applications. The proposed logic gate is also capable of implementing all Boolean logic gates in either one or two steps. The gate implements the NOR and NAND functions directly. This is coupled with the standard IMPLY NOT function to create a logically complete set.

The proposed logic gate has a very similar circuit design to the IMPLY gate but uses a third memristor. This third memristor is used in order to be able to perform the NAND and NOR operation in a single step and store the result in a separate memristor, without overwriting either input memristor. Figure 135 shows the logic circuit for the proposed gate.

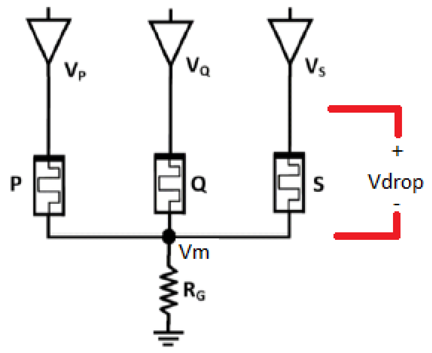


Figure 135: Proposed memristor logic circuit

The IMPLY NAND operation requires 2 steps and 3 memristors and the NOR operation requires 3 steps and 4 memristors. The proposed gate requires 1 step and 3 memristors for each. Table 49 shows the voltages applied in each execution step for a NAND and NOR operation in the IMPLY and proposed domains.

Table 49: Voltages applied for NAND and NOR operations in IMPLY and proposed domains

NAND	**IMPLY**			**Proposed**		
	Vp	**Vq**	**Vs**	**Vp**	**Vq**	**Vs**
1		V_{cond}	V_{set}	V_{cond}	V_{cond}	V_{set}*
2	V_{cond}		V_{set}			

NOR	**IMPLY**				**Proposed**		
	Vp	**Vq**	**Vr**	**Vs**	**Vp**	**Vq**	**Vs**
1	V_{cond}		V_{set}		V_{cond}	V_{cond}	V_{set} t+
2		V_{set}	V_{cond}				
3		V_{cond}		V_{set}			

Here, $V_{cond} = 1.6V$ and $V_{set} = 2.5V$. The proposed gates use the same fundamental reasoning and calculations to resolve the V_{set} signals as the IMPLY gate. Let the common node between each of the operands be V_m and let the output memristor S be initialized to 0 (high resistance). The circuit is essentially a voltage divider at V_m. Thus, for the proposed circuit in Figure 135, if high resistance=100K ohms and low resistance=1K ohms, the voltage at V_m is as shown in Table 50.

Table 50: Voltage characteristics of the proposed circuit for V_{cond}=1.6 V

Input {PQ}	V_m	V_{dropS}	V_{dropS} for V_{set} =2.85V	V_{dropS} for V_{set} =2.6V
00	$.06 + V_{set}/53$	$52\, V_{set}/53-.06$	2.74 V	2.49 V
01	$1.06+ V_{set}/152$	$151\, V_{set}/152-1.06$	1.77 V	1.52 V
10	$1.06+ V_{set}/152$	$151\, V_{set}/152-1.06$	1.77 V	1.52 V
11	$1.275+ V_{set}/251$	$250\, V_{set}/251-1.275$	1.56 V	1.315 V

V_{set} now represents the value applied to the S memristor (either V_{set}* or V_{set}+ for the NAND and NOR operation) and V_{dropS} represents the voltage across the S memristor.

For the NAND operations, the result memristor S should only be set for inputs 00, 01, and 10. Thus in Table 50, the first three rows in the V_{dropS} column must be greater than the V_{th} of the memristor to change its value and the fourth row must be less. Specifically, $(250V_{set}*/251)-1.275<V_{th}<(151V_{set}*/152)-1.06$. For this work, Let $V_{th}=V_{cond}=1.6V$. Thus, $2.68V < V_{set}* < 2.8865V$. Let $V_{set}* = 2.85V$. The voltage drops across the result are shown in the third column. Notice that the voltage drop is above the threshold for cases 00, 01, and 10, but below the threshold for 11. Thus, the result memristor will be set in the first three cases, implementing the NAND operation. So, to perform a NAND operation, $V_{cond}=1.6V$ is applied to memristors P and Q and $V_{set}* = 2.85V$ is applied to memristor S.

A similar methodology is performed for the NOR operation. For the NOR operation, only the input case 00 should result in a '1' on the output memristor. Thus, $(151V_{set}+/152)-1.06 < V_{th} < (52V_{set}+/53)-.06$ or $1.69V < V_{set}+ < 2.68V$. Let $V_{set}+ = 2.6V$. The results for the drop across the result memristor are shown in the final column of the table. Again, the correct functionality is achieved. So, to perform a NOR operation $V_{cond}=1.6V$ is applied to memristors P and Q and $V_{set}* = 2.6V$ is applied to memristor S.

By coupling the NOR and NAND operation with the traditional IMPLY NOT operation, a complete Boolean logic set is established. The standard IMPLY NOT is chosen since it already requires the minimal area and delay.

This logic gate can be incorporated into a crossbar in the exact same manner as an IMPLY gate. The only difference is the additional variant V_{set} values to allow each operation to perform. For standard IMPLY operations, the voltages V_{reset}, V_{cond}, and V_{set}

are required. For the proposed work the voltages V_{reset}, V_{cond}, V_{set}*, and V_{set}+ per operation are required. However, the work reduces the number of steps for NAND and NOR to 1 and AND and OR to 2. This also reduces power consumptions since voltages are now applied for fewer steps. Also, the complexity is reduced to 1 memristor per operand for the proposed approach. This logic gate requires fewer steps than the MAD gate crossbar implementation and does not require threshold detection but requires more application voltages. It also has delay competitive with alternative approaches while offering lower delay and a more complete set of operations.

This logic gate can also be extended to N-bit operations by simply changing the value of V_{set}. A V_{cond} signal is applied to all of the input memristors P0-P7 in parallel, while the output memristor S is applied a V_{set} voltage which can be calculated in the same manner. For example, for an 8-bit NOR, V_{set} must satisfy $1.83V < V_{set} < 2.726V$. Thus, the same V_{set} as used for the original NOR gate, V_{set}=2.6V can be maintained even for increasing operation widths. This is especially useful for performing logical operations on 2 N-bit operands in parallel within memory. No other proposed logic gates have shown this functionality.

PROPOSED FUTURE WORK

Given the infancy of memristor research, there are many avenues for future directions in this area.

First and foremost, improved simulation models and understanding of memristors are critical to the advancement of their use. Without introducing them to the academic community and university students, memristors will not become a commonplace in logic design. One catalyst for this is to establish accurate, available, easy-to-use models for memristors. This would allow students, researchers, and industry to explore memristor behaviors and applications at a more rapid, progressive, rate. This would in turn lead to more ideas and faster developments and design cycles for memristor designs. Ideally, power and energy measurements would be incorporated into these models. Additionally, there is currently no model for GOTO pairs that can be used in simulation or design. Establishing one would enable researchers to simulate and measure nanoscale designs like the threshold gate designs presented in this dissertation.

Secondly, research can focus on the application of memristors to the remainder of modern system architecture. As the complexity of the system increases, there are more opportunities that could be leveraged from memristors, especially when you consider the fact that their logic can be pipelined much more efficiently and at a finer-granularity than traditional CMOS. Research into the impact of this new way of thinking about dataflow should be further investigated.

Throughout this dissertation, the main application of the memristors was on the actual logic structures that store and transfer data. However, another interesting approach is to consider the use of memristors for the control logic. For example, consider if the memristor weights in the threshold gate designs could be dynamically modified during execution. It is possible to change these resistance weights because of memristor characteristics. Thus, nothing precludes these values from changing between operations. This would allow for logic reuse and reprogrammable behavior for various logic operations. Similarly, this dissertation discussed that MAD gates must set appropriate threshold voltages on switches in order to perform desired operations. Perhaps these thresholds could also dynamically change. Research into the design of these types of modules would bridge the gap between customizable and efficient hardware, offering both.

Lastly, a large majority of the research that has been performed on larger scale memristor designs has been in the simulation domain. No implementations have been run and measured on real memristors. This slows down the community in fully understanding and accepting memristors as a design component. Real-world measurements would give insight into realistic limitations that ideal models do not capture in simulation, as well as help drive the models and simulations being designed.

CONCLUSION

In this dissertation, novel and improved memristor-based designs for arithmetic units are presented. MAD logic is presented as a novel lower complexity, lower delay alternative to prior approaches at memristor-based Boolean logic. MAD gates are standardized, each requiring only 1 memristor per operand and a single step for execution. The MAD gates, along with three other approaches - IMPLY, hybrid-CMOS, and threshold gates - are then extended to numerous adder, multiplier, and divider designs. The proposed MAD gates create designs with fewer components and lower step delay than all other approaches.

The designs are analyzed in terms of delay and complexity and compared against each other, prior work, and traditional CMOS designs. The proposed adders, multipliers, and dividers were heavily optimized to improve in both measures (complexity and delay) over the traditional CMOS implementations. As intermediate steps to these arithmetic units, their building blocks are also implemented using memristors. For example, full adders, XOR gates, multiplexers, modified half adders, and shift registers were designed, optimized, and analyzed. This is the first known work to explore these units in IMPLY, hybrid-CMOS, threshold gate, and MAD gate domains.

The contributions of this dissertation confirm the potential for performance and complexity improvements for memristor-based arithmetic units. All implemented approaches improve upon their CMOS counterparts in complexity, delay, or both. MAD gates stand out as the lowest delay, lowest complexity option by a significant margin for nearly every arithmetic unit designed in this work. The MAD gates offer further benefits by amortizing latency and increasing throughput via pipelining operations through the memristor circuitry. Although each approach has tradeoffs and benefits that make them optimal for given applications, MAD gates offer the general best-case option for memristor-based logic.

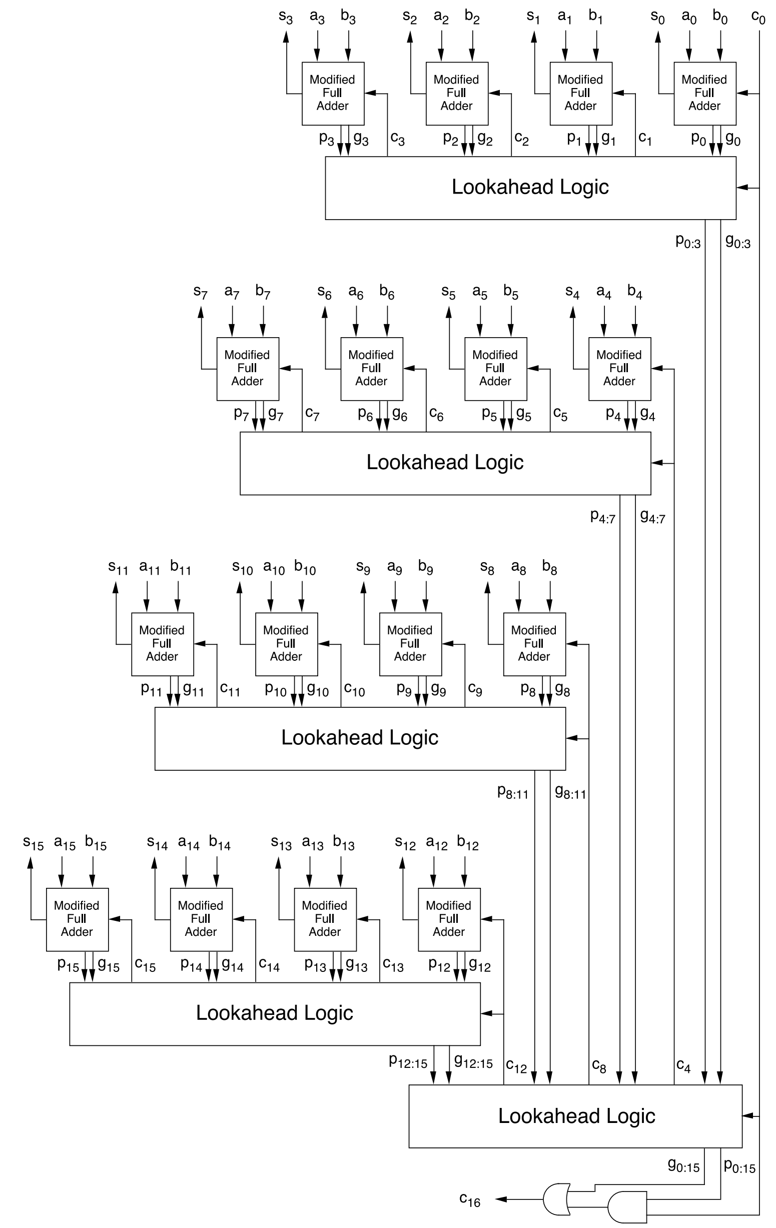
The work in this dissertation motivates further work in the exploration of full-system memristor implementations. Modern architecture should be re-envisioned from the top level down, applying the findings in this work to the rest of the system. The key takeaways from this work, namely MAD gates, the low complexity and delay of memristor-based designs, and pipelined memristor circuitry, can be leveraged to design entirely novel systems that break the limits of traditional CMOS design.

Appendix A: Full List of Parameters for the TEaM Memristor Model

Table A.1: List of selected parameters for the TEaM memristor model

Parameter	Value
Rhigh	100K
Rlow	1K
D	3e-9
uv	1e-15
w multiplied	1e9
p coefficient	2
J	1.5
window noise	1e-18
threshold voltage	0
Coff	3.5e-6
Con	4e-5
Ioff	20e-6
Ion	-20e-6
Xc	107e-11
B	0
Aon	1.8e-9
Aoff	1.2e-9
Kon	-1
Koff	10
AlphaOn	1
AlphaOff	1
Von	-1.78
Voff	0.0115
IV relation	1
Xon	3e-9
Xoff	0

Appendix B: 16-bit Carry Lookahead Adder Schematic



Modified Full Adder
Lookahead Logic
s_0 a_0 b_0 c_0 p_0 g_0 c_1
s_1 a_1 b_1 p_1 g_1 c_2
s_2 a_2 b_2 p_2 g_2 c_3
s_3 a_3 b_3 p_3 g_3
$p_{0:3}$ $g_{0:3}$
s_4 a_4 b_4 p_4 g_4 c_5
s_5 a_5 b_5 p_5 g_5 c_6
s_6 a_6 b_6 p_6 g_6 c_7
s_7 a_7 b_7 p_7 g_7
$p_{4:7}$ $g_{4:7}$
s_8 a_8 b_8 p_8 g_8 c_9
s_9 a_9 b_9 p_9 g_9 c_{10}
s_{10} a_{10} b_{10} p_{10} g_{10} c_{11}
s_{11} a_{11} b_{11} p_{11} g_{11}
$p_{8:11}$ $g_{8:11}$
s_{12} a_{12} b_{12} p_{12} g_{12} c_{13}
s_{13} a_{13} b_{13} p_{13} g_{13} c_{14}
s_{14} a_{14} b_{14} p_{14} g_{14} c_{15}
s_{15} a_{15} b_{15} p_{15} g_{15}
$p_{12:15}$ $g_{12:15}$ c_{12} c_8 c_4
$g_{0:15}$ $p_{0:15}$
c_{16}

Figure B.1: 16-bit carry lookahead adder diagram

Appendix C: 8-bit Hybrid-CMOS Conditional Sum Adder Schematic

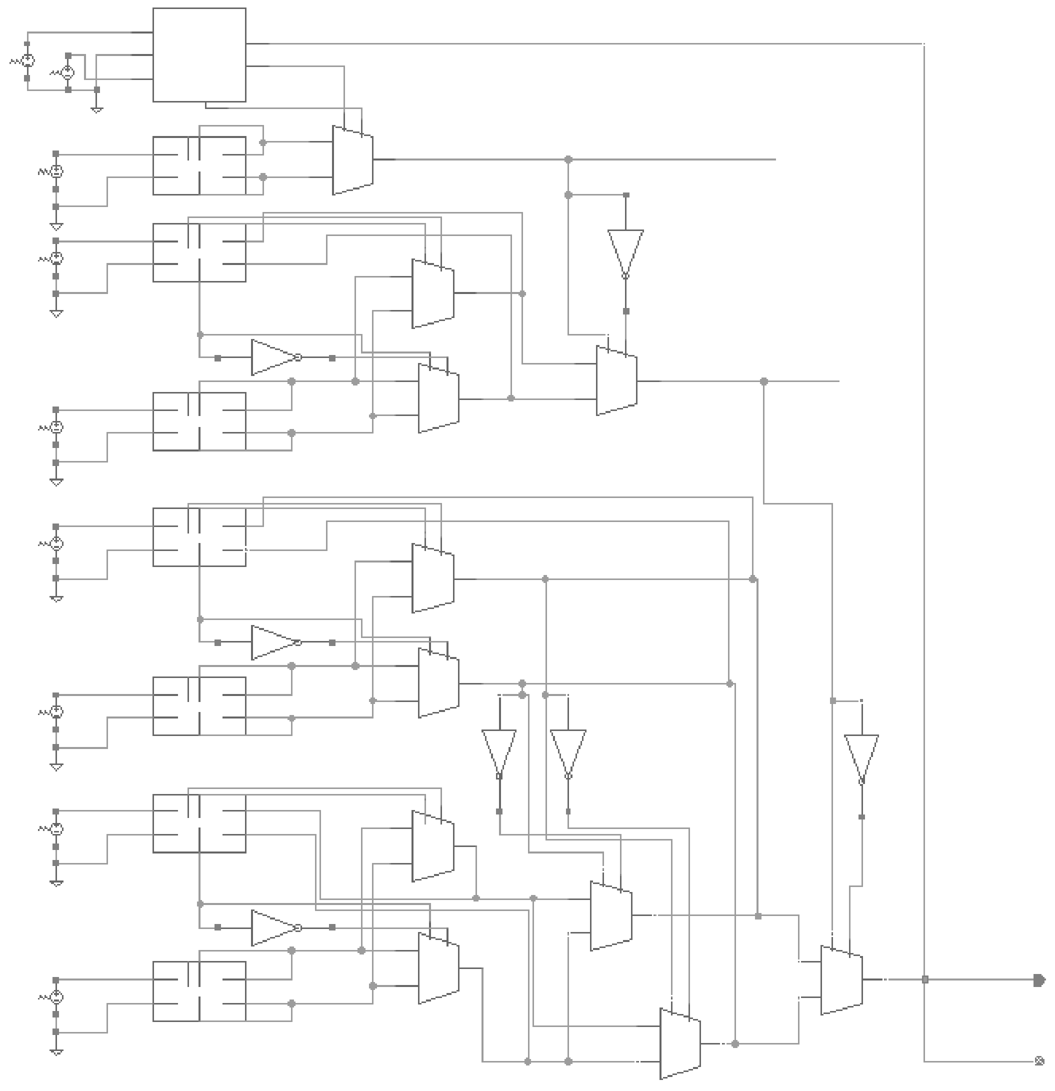


Figure C.1: 8-bit hybrid-CMOS conditional sum adder schematic

Appendix D: 4-bit Hybrid-CMOS SRT Divider Schematic

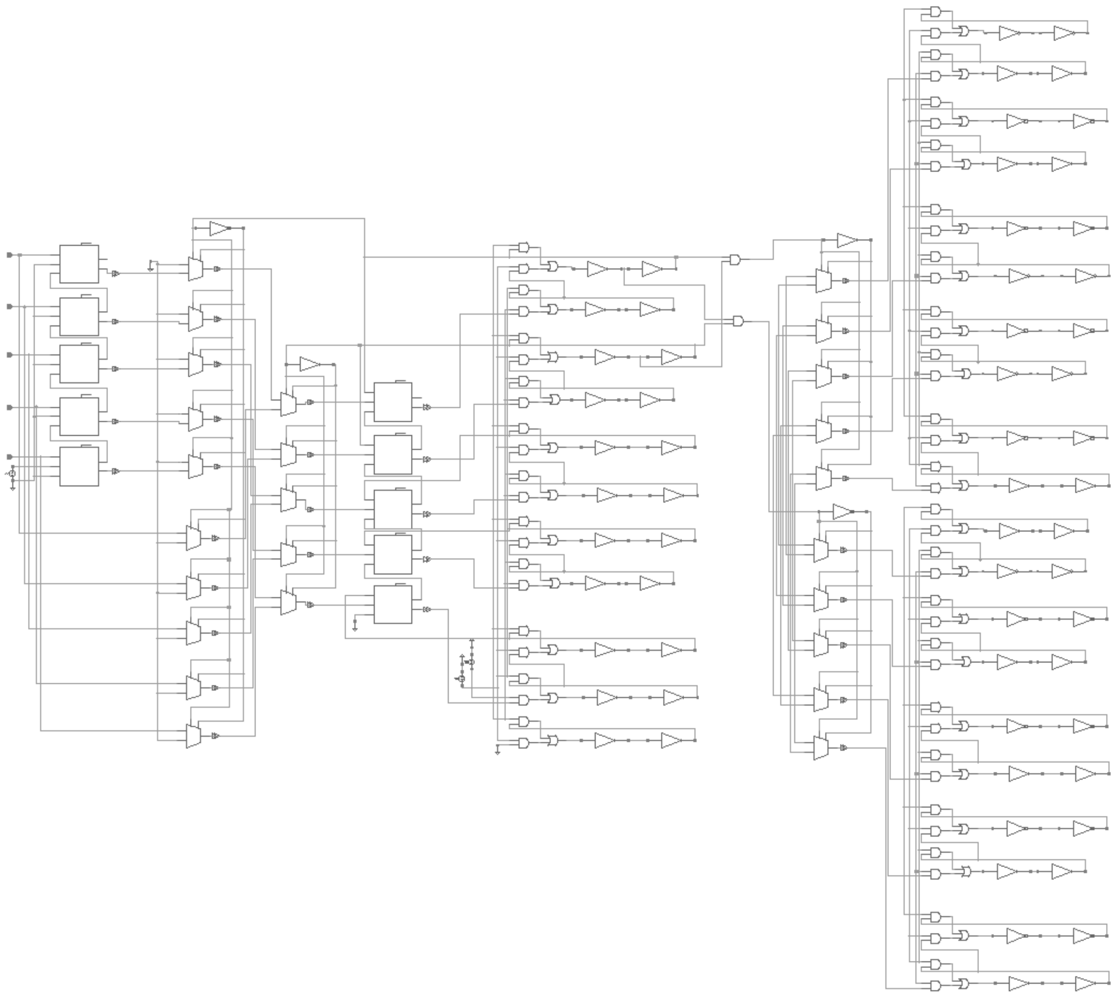


Figure D.1: 4-bit hybrid-CMOS SRT divider schematic

Appendix E: Complete List of Publications

F. Zhou, L. Guckert, Y. Chang, E. E. Swartzlander Jr., and J. Lee, “Bidirectional voltage biased implication operations using SiOx based unipolar memristors,” *Applied Physics Letters,* Nov. 2015.

L. Guckert and E. E. Swartzlander, Jr., "MAD Gates - Memristor Logic Design Using Driver Circuitry," *IEEE Transactions on Circuits and Systems II: Express Briefs*, Vol. 99, pp.1, Apr. 2016.

L. Guckert and E. E. Swartzlander Jr., “Implementing Adders with Memristors,” *ETCMOS Conference*, May 2016.

L. Guckert and E. E. Swartzlander Jr.,*“Memristor-based Threshold Gate Adders,” DAC Conference,* Jun. 2016.

L. Guckert and E. E. Swartzlander Jr., “Optimized Memristor-Based Multipliers,” *IEEE Transactions on Circuits and Systems I: Regular Papers,* Oct. 2016.

L. Guckert and E. E. Swartzlander Jr., “Modern System Design using Memristors,” Chapter in *Advances in Memristors, Memristive Devices and Systems,* Springer-Verlag, Oct. 2016.

L. Guckert and E. E. Swartzlander Jr., “Carry Lookahead Adder Design Using Memristors,” Chapter in *Devices, Circuits and Systems,* CRC Press, Oct. 2016.

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].

L. Guckert and E. E. Swartzlander Jr., “Memristor-based Logic Design for Dadda Multipliers,” *ETCMOS Conference,* May 2017.

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] T. Devolder, J. Hayakawa, K. Ito, H. Takahashi, S. Ikeda, P. Crozat, N. Zerounian, J.-V. Kim, C. Chappert, and H. Ohno, "Single-Shottime-Resolved Measurements of Nanosecond-Scale Spin-Transfer Induced Switching: Stochastic Versus Deterministic Aspects," *Physical Review Letters*, Vol. 100, pp. 057206, 2008.

[5] 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.

[6] E. Lehtonen and M. Laiho, "CNN Using Memristors for Neighborhood Connections," *12th International Workshop on Cellular Nanoscale Networks and Their Applications (CNNA),* pp. 1-4, Feb. 2010.

[7] S. Kvatinsky, E. G. Friedman, A. Kolodny, and U. C. Weiser, "TEAM: ThrEshold Adaptive Memristor Model," *IEEE Transactions on Circuits and Systems I: Regular Papers,* Vol. 60, pp. 211-221, Jan. 2013.

[8] Z. Biolek, D. Biolek, and V. Biolkova, "SPICE Model of Memristor with Nonlinear Dopant Drift," *Radioengineering*, Vol. 18, pp. 210–214, June 2009.

[9] M. D. Pickett, D. B. Strukov, J. L. Borghetti, J. J. Yang, G. S. Snider, D. R. Stewart, and R. S. Williams, "Switching Dynamics in Titanium Dioxide Memristive Devices," *Journal of Applied Physics*, Vol. 106, pp. 1–6, Oct. 2009.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] K. Bickerstaff and E. E. Swartzlander, Jr., "Memristor-Based Arithmetic," *Conference Record of the Forty Fourth Asilomar Conference on Signals, Systems and Computers*, pp. 1173-1177, Nov. 2010.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] S. Kvatinsky, N. Wald, G. Satat, A. Kolodny, U. C. Weiser, U.C. and E. G. Friedman, "MRL — Memristor Ratioed Logic," *13th International Workshop on*

Cellular Nanoscale Networks and Their Applications (CNNA), pp. 1-6, Aug. 2012.

[21] T. Singh, "Hybrid Memristor-CMOS (MeMOS) based Logic Gates and Adder Circuits," *CoRR*, pp. 1-11, June 2015.

[22] 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].

[23] G. Rose, J. Rajendran, H. Manem, R. Karri, and R. Pino, "Leveraging Memristive Systems in the Construction of Digital Logic Circuits," *Proceedings of the IEEE,* Vol. 100, pp. 2033-2049, June 2012.

[24] C. Collier, J. Jeppesen, Y. Luo, J. Perkins, E. Wong, J. Heath, and J. Stoddart, "Molecular-based Electronically Switchable Tunnel Junction Devices," *Journal of the American Chemical Society*, Vol. 123, pp. 632–641, Dec. 2001.

[25] G. Rose and M. Stan, Jr., "A Programmable Majority Logic Array Using Molecular Scale Electronics," *IEEE Transactions on Circuits and Systems I: Regular Papers,* Vol. 54, pp. 2380–2390, Nov. 2007.

[26] E. Goto, K. Murata, K. Nakazawa, K. Nakagawa, T. Moto-Oka, Y. Ishibashi, T. Soma, and E. Wada, "Esaki Diode High-Speed Logical Circuits," *IRE Transactions on Electronic Computers*, Vol. EC-9, pp. 25–29, Mar. 1960.

[27] T. Huisman, "Counters and Multipliers with Threshold Logic," M.S. thesis, Delft University of Technology, May 1995.

[28] J. Zhou, Y. Tang, J. Wu, X. Fang, X. Zhu, and D. Huang, "Design of Counters Based on Memristors," *Proceedings of the 2013 International Conference on Information System and Engineering Management*, ICISEM 2013.

[29] L. Guckert and E. E. Swartzlander, Jr., "MAD Gates - Memristor Logic Design Using Driver Circuitry," *IEEE Transactions on Circuits and Systems II: Express Briefs*, Vol. 99, pp.1, Apr. 2016.

[30] F. Corinto and A. Ascoli, "A Boundary Condition-based Approach to the Modeling of Memristor Nanostructures," *IEEE Transactions on Circuits and Systems I: Regular Papers*, Vol. 59, pp. 2713–2726, Nov. 2012.

[31] Y. Deng, P. Huang, B. Chen, X. Yang, B. Gao, J. Wang, L. Zeng, G. Du, J. Kang, and X. Liu, "Rram Crossbar Array with Cell Selection Device: A Device and Circuit Interaction Study," *IEEE Transactions on Electron Devices*, Vol. 60, pp. 719–726, 2013.

[32] C. Babbage, "Passages from the Life of a Philosopher," Longman, Green, Longmand Roberts & Green, pp. 59-63, 114-116, 1864.

[33] F. Merrikh-Bayat and S. Bagheri Shouraki, "Memristor-based Circuits for Performing Basic Arithmetic Operations," Computing Research Repository, 2010.

[34] M. D. Ercegovac and T Lang, "On-the-Fly Conversion of Redundant into Conventional Representations," *IEEE Transactions on Computing,* Vol. 36, pp. 895-897, 1987.

[35] 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.

[36] P. Celinski, J. F. López, S. Al-Sarawi, and D. Abbott, "Low Depth, Low Power Carry Lookahead Adders Using Threshold Logic," *Microelectronics Journal*, Vol. 33, pp. 1071-1077, Dec. 2002.

Vita

Lauren Guckert was born and raised with her family in Austin, Texas. Lauren attended The University of Texas at Austin and studied Electrical and Computer Engineering, receiving her B.S. degree in 2012. She also received her M.S. degree in Electrical and Computer Engineering with a concentration in Computer Architecture and Embedded Processors from The University of Texas at Austin in 2015. Shortly after, she began working at ARM Inc. in Austin, TX as an engineer for the performance modeling and analysis team. She received her Ph.D. degree in Electrical and Computer Engineering from The University of Texas at Austin in December 2016 for her research on low-area, low-delay, and low-power memristor-based logic with applications for arithmetic and logic-in-memory.

Lauren Guckert can be contacted at her permanent email, lguckert@gmail.com

This dissertation was typed by the author.