Spring 6-6-2016

Complete Design Methodology of a Massively Parallel and Pipelined Memristive Stateful IMPLY Logic Based Reconfigurable Architecture

Kamela Choudhury Rahman
Portland State University

Complete Design Methodology of a Massively Parallel and Pipelined Memristive Stateful IMPLY Logic Based Reconfigurable Architecture

by

Kamela Choudhury Rahman

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Electrical and Computer Engineering

Dissertation Committee:
Marek A Perkowski, Chair
Dan Hammerstrom
Xiaoyu Song
Rolf Koenenkamp

Portland State University
2016

Abstract

Continued dimensional scaling of CMOS processes is approaching fundamental limits and therefore, alternate new devices and microarchitectures are explored to address the growing need of area scaling and performance gain. New nanotechnologies, such as memristors, emerge. Memristors can be used to perform stateful logic with nanowire crossbars, which allows for implementation of very large binary networks that can be easily reconfigured. This research involves the design of a memristor-based massively parallel datapath for various applications, specifically SIMD (Single Instruction Multiple Data) like architecture, and parallel pipelines. The dissertation develops a new model of massively parallel memristor-CMOS hybrid datapath architectures at a systems level, as well as a complete methodology to design them. One innovation of the proposed approach is that the datapath design is based on space-time diagrams that use stateful IMPLY gates built from binary memristors. This notation aids in the circuit minimization in logic design, calculations of delay and memristor costs, and sneak-path avoidance. Another innovation of the proposed methodology is a general, new, architecture model, MsFSMD (Memristive stateful Finite State Machine with Datapath) that has two interacting sub-systems: 1) a controller composed of a memristive RAM, MsRAM, to act as a pulse generator, along with a finite state machine realized in CMOS, a CMOS counter, CMOS multiplexers and CMOS decoders, 2) massively parallel, pipelined, datapath realized with a new variant of a CMOL-like nanowire crossbar array, MsCMOL (Memristive stateful CMOL), with binary stateful memristor-based IMPLY gates. Next contribution of the dissertation is the new type of FPGA. In contrast to the previous memristor-based FPGA (mrFPGA), the

proposed MsFPGA (Memristive stateful logic Field Programmable Gate Array) uses memristors for memory, connections programming, and combinational logic implementation. With a regular structure of square abutting blocks of memristive nanowire crossbars and their short connections, proposed architecture is highly reconfigurable. As an example of using the proposed new FPGA to realize biologically inspired systems, the detailed design of a pipelined Euclidean Distance processor was presented and its various applications are mentioned. Euclidean Distance calculation is widely used by many neural network and associative memory based algorithms.

Dedication

To my children, Favian and Nashita

Acknowledgments

I would like to thank my adviser, Professor Marek Perkowski, with the Department of Electrical and Computer Engineering, Portland State University, for his invaluable time and relentless effort in mentoring me during my journey to finish my Ph.D. research and dissertation. I not only learned how to define and solve an engineering problem, but also to think critically and in innovative ways. With his vast knowledge on advanced logic synthesis and architecture, he continuously challenged me with probing questions and directed me in producing high quality work. I also learned how to write about technically complex topics, a skill I know will be valuable to me in the future. Professor Perkowski's dedication to education and serving others is a true inspiration for me.

I would also like to express my deep-felt gratitude to my former adviser and current committee member Professor Dan Hammerstrom, also with the Department of Electrical and Computer Engineering at Portland State University, for giving me the opportunity to work with his research team and introducing me to the amazing world of biologically inspired computation. My prior profession as a circuit designer motivated me to find hardware solutions for those biologically inspired algorithms, and thus later became the topic of this dissertation. From Professor Hammerstrom, I learned a lot by observing his consistently calm, non-reactive and patient personality. I am especially indebted to Professor Hammerstrom for his continued guidance during his tenure at DARPA in Washington D.C.

I must admit that without Professor Hammerstrom's persistent support and confidence in my potential, and without Professor Perkowski's relentless guidance, it would not be possible for me to complete my doctoral dissertation.

I also wish to thank the other members of my committee, Professor Xiaoyu Song of the Department of Electrical and Computer Engineering and Professor Rolf Koenenkamp of the Department of Physics, both with Portland State University. Their suggestions, comments and additional guidance were invaluable.

I would like to thank Professor C. Teuscher of the Department of Electrical and Computer Engineering, Portland State University for supervising the NSF grant in the year 2013. I also appreciate the time of Professor D. B. Strukov of the Department of Electrical and Computer Engineering, University of California, Santa Barbara and Professor Wei Lu of the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor for helpful discussions on nanowire crossbar simulations and memristor-nanowire power calculations.

I would also like to thank several former and current peer students in the department of Electrical and Computer Engineering at Portland State University, Professor M. Zaveri, D. Voils, Hongayn Xiyao, B. Ryu, and Yiwei Li for many helpful discussions.

Finally, I would like to introduce my family and their contributions. Many years back, holding my father's hand, my journey to Kindergarten began. Today my father is watching over me from heaven. Although my mother did not get the opportunity to pursue her own education, she was quite ambitious about the success of my career. Since the days of my undergraduate studies, my husband has provided me with enormous support, inspiration and motivation. My son Favian always wanted me to have a great

career. It made him upset when few years back I left a position from a top US corporation, so that I could spend more time with him and his younger sister Nashita. Both of them constantly reminded me of a successful completion of my doctoral work so that they could take pride in my success. My two older brothers are my other constant well-wishers. I am grateful to my aunt for her many contributions since my childhood. I am indebted to my numerous teachers in every step of my education for their contributions. I am grateful to my friends and family from across the globe, who constantly encouraged me to successfully complete my PhD. I also recognize my cat Tucker. His image is used to explain the proposed methodology in the dissertation.

Lastly, I thankfully acknowledge Professor Hammerstrom for funding my research through the National Science Foundation NSF-CDI under award #1028378.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

The following table describes the significance of various abbreviations and acronyms used throughout the dissertation. The page on which each one is defined or first used is also given.

Abbreviation	Meaning	Page
AM	Associative Memory	1
CMOL	hybrid CMOS/MOLecular device at nanowire intersections	3
ED	Euclidean Distance	4, 43
FPGA	Field Programmable Gate Array	11, 33
FSMD	Finite State Machine with Datapath	34-36
IM	Input Memristor	21
MsCMOL	Memristive stateful CMOL	14, 32, 85, 89-90
MsFPGA	Memristive stateful logci FPGA	11, 33-35
MsFSMD	Memristive stateful FSMD	34-36
MsRAM	Memristive stateful RAM	35-43
NVM	Non Volatile Memory	87
NW	Nanowire	170
PI	Primary Input	98
SIMD	Single Instruction Multiple Data	45
WM	Working Memristor	21

1 INTRODUCTION

1.1 Research Outline

1.1.1 Neural Network or Biologically Inspired Modeling

By definition, any system that tries to model the architectural details of the neocortex is a biologically inspired model or neural network [54][55]. Computers cannot accomplish human-like performance for many tasks such as visual pattern recognition, understanding spoken language, recognizing and manipulating objects by touch, and navigating in a complex world. After decades of research, there exist no significant viable algorithms to achieve human-like performance on a computer or special hardware accelerator. So far, there has not been much research and development in hardware for the biologically inspired software models. The hardware implementation of large-scale neural networks is an excellent candidate application for the high density computation and storage possible with current and emerging semiconductor technologies [84]. Besides, hardware implementation is much faster than software, the primary motivation for this dissertation research is to engineer a system level design in hardware that can be used for many biologically inspired computation and other similar applications.

1.1.2 Associative Memory

An associative memory (AM) [50] can recall information from incomplete or noisy inputs and as such, AM has applications in pattern recognition, facial recognition, robot

vision, robot motion, DSP, voice recognition, and big data analysis. Research on the potential mapping of the AMs onto the nano-scale electronics provides useful insight into the development of non-von-Neumann neuromorphic architectures. A datapath for implementing an AM can be implemented using common hardware elements, such as, adder, multiplier, simple divider, sorter, comparator and counter.

Therefore, providing a methodology for non-von-Neuman architecture with nanoscale circuits and devices is one of the targets of this research.

1.1.3 Massively Parallel Architecture

Neural network based algorithms generally require massive parallelism. Single Instruction Multiple Data (SIMD) [95], pipelining, and systolic array architecture [95] are typical to DSP, neural network and image processing algorithms.

The goal of this research is to propose a design methodology for a complete system that can handle large number of wide vectors with a series of SIMD (Single Instruction Multiple Data)-like processing elements and pipelined architecture.

1.1.4 Neuromorphic Circuits and Devices

The emergence of many novel nanotechnologies has been primarily driven by the expected scaling limits in conventional CMOS processes. Through such efforts many new and interesting novel neuromorphic circuits and devices have been discovered and invented. Memristor is an example of such a new technology.

A memristor feature size of F = 50 nm (where, *F* is the lithographic feature size or half-pitch i.e. half of center-to-center nanowire distance) yields a synaptic density of 10^{10} memristive synapses per square centimeter, which is comparable to that of the human cortex [89][90]. Therefore, memristor technology shows the prospect of scaling up the capacities of DSP and Image Processing architectures, and associative memories. Hybrid CMOS-Memristor design could be used for architectures which due to their complexity cannot be designed and simulated in real-time in hardware-software using conventional CMOS based design.

As such, this research undertakes the implementation of a complete system level design using binary memristors with IMPLY logic and using a new variant of a CMOL crossbar nano-grid array, MsCMOL (Memristive stateful CMOL). CMOL is defined as a two-layer hybrid technology, in which semiconductor CMOS transistors are placed in the lower layer, and molecular scale two-layer two-terminal nanodevices are placed at the upper layer [Strukov-Likharev, 2005].

1.1.5 Design Methodology Development

The essence of this dissertation work is to develop a new methodology to design a massively parallel and pipelined architecture at a system level using binary memristors for biologically inspired Associative Memory and other similar application areas as mentioned before. The research proposed here will involve the design of an IMPLY-memristor based massively parallel reconfigurable architecture at a system and logic levels.

1.2 Research Background and Motivation

1.2.1 Part 1: Research Groundwork

1.2.1.1 Defining Associative Memory

Associative memory (AM) [53][62] is a system that stores mappings from input representations to output representations. When the input pattern is given, the output pattern can be reliably retrieved. When the input is incomplete or noisy, the AM is still able to return the output result corresponding to the original input based on a *Best Match* procedure where the memory selects the input vector with the closest match, assuming some metric, to the given input, then returns the output vector for this closest matched input vector.

In *Best Match* associative memory, vector retrieval is done by matching the contents of each location to a key. This key could represent a subset or a corrupted version of the desired vector. The memory then returns the vector that is *closest* to the key. Here, *closest* is based on some metric, such as ***Euclidean Distance*** [19][36][37][38][39][40][41][42][43][44][45]. Likewise, the metric can be conditioned so that some vectors are more likely than others, leading to Bayesian-like inference.

As in associative memories the information is retrieved through a search: given an input vector one wants to obtain the stored vector that has been previously associated with the input. In a parallel hardware implementation of a large-scale associative memory the memory is searched to find the minimum distance between the new vector and the stored memory vector using the Euclidean distance formula.

On the other hand, the *Exact Match* association, as in the traditional content addressable memory (CAM), returns the stored value that corresponding to the exactly matched input. A CAM holds a list of vectors which are distinguished by their addresses, when a particular vector is needed, the exact address of the vector must be provided.

1.2.1.2 History of Associative Memory Algorithm Development

Associative memories can be of different types. The first associative memory model called *Die Lernmatrix* was introduced by Steinbuch and Piske in 1963. Willshaw modeled and modified versions (1969-1999) [53], Palm developed a version (1980) [73], and an iterative Palm model (1997). The Brain-state-in-a-box (BSB) was developed by Anderson et al. (1977, 1993, 1995, 2007). Likewise there is the Hopfield network model (1982) [64], the Self-Organizing Map (SOM) proposed by Kohonen (1982, 1989) [76][82], the Dynamical Associative Memory (DAM) by Amari (1988, 1989), Bidirectional Associative Memory (BAM) by Kosko (1988) [68], Sparse Distributed Memory (SDM) by Kanerva (1988) [56][65][66], Bayesian Confidence propagation Neural Network (BCPNN) by Lansner et al. (1989) [75], Cortronic networks by Hecht-Nielsen (1999), and Correlation matrix memories (CMM) [77]. Furber developed his own model (2007) implemented using Spiking Neural Networks (SNN) [51][52][60][72], and there are, Spatial and Temporal versions of Self-Organizing Incremental Neural Networks (SOINN, ESOINN, GAM) by Shen and

Hasegawa (2006-2011) [57][79][80]. Finally, the Cortical Learning Algorithms (CLA) developed by Numenta (2010) are examples of some associative memories [61].

1.2.1.3 System Input Data Encoding

The input data into a neural network system can be received in any form, e.g. binary data, real-valued data etc. Input data then gets encoded as wide vectors. Different neural network models follow different encoding mechanism.

After generating the vectors, the similarity between the vectors is measured using the Euclidean distance calculation or calculating the dot product of the two vectors. The similarity is measured through a *distance threshold* value. A distance threshold constant is used to control the classification of a new node to a new class or to an existing class. During the experimentation, the values of distance threshold are changed several times. A small value of distance threshold may result in a large number of classes. With further experimentation, it is possible to obtain even fewer classes at the output by iterating on the distance threshold constant.

1.2.1.4 Evaluation of Associative Memory Algorithm

The purpose of the research was to provide hardware directions for biologically inspired associative memory models. Many groups have developed biologically inspired software based algorithms [61][79][80] to address such problems. A few groups are looking into creating increasingly more sophisticated hardware based models of neural circuits [63][67][87][88][89][93][96], and then apply those models to real applications.

Finding a suitable associative memory algorithm was the initial task for this dissertation work. Through a detailed literature search, some of the most promising models were identified. First, the performance of the associative memory model was evaluated. Next, the capability of sequential or temporal pattern prediction was checked. Based on all the results published by other authors and my own experimentation with software models, one suitable model was identified for this research.

As a part of this dissertation work, the Kanerva (Furber) SDM Associative memory model and the Palm Associative memory models were implemented in Matlab. After evaluation of the two models using the same datasets [69], it was not possible to prove the superiority of one model over the other, as both models showed some capability as well as some inaccuracies. The CLA Model [61] used for these experiments is a commercial model by Numenta, Inc. 2010. The Furber Model is a model that was coded in Matlab as a part of my dissertation research, and the coding required certain assumptions based on the original published work by Furber et al. [51][52]. In addition, although the CLA model has the *variable order sequence prediction* feature [61], the experimental results did not show performance superiority of the CLA model over the Furber Model. As such, we were unable to justify that the CLA model is performing any better than the Furber model and we concluded that both models have similar performance and none of the models are completely error free.

These conclusions were the motivation behind an additional literature search to find more models that can provide better solutions to the problem. A more promising biologically inspired associative memory model for spatial and temporal pattern

recognition by Shen and Hasegawa [57][79][80] was found through further literature search. This led to the study of their SOINN model, the ESOINN Model and finally the GAM model, which is the most promising algorithms among all of the models studied. For the purpose of this research, the SOINN [57], and ESOINN algorithm [79] were coded in Matlab for *spatial pattern recognition*. Later, the complete GAM [80] algorithm was also coded [Appendix A], the algorithm does both spatial and temporal pattern recognition. For the *spatial pattern recognition* experiments, input data was collected from Lecun's MNIST hand-written digit database [70] both for training and test purposes. Upon completion of the training, a different set of images were used to test the performance of the algorithm.

1.2.1.5 ESOINN/ GAM

Shen and Hasegawa proposed several models on pattern recognition, such as the Self-Organizing Incremental Neural Network (SOINN) [57] based on an unsupervised learning technique [58], and the Enhanced Self-Organizing Incremental Neural Network (ESOINN) [79], which is a modification of SOINN. Both of these algorithms have applications in spatial pattern recognition. Shen and Hasegawa also published a General Associative Memory (GAM) algorithm [80], which is an associative memory based algorithm, and a temporal version of the SOINN algorithm. The GAM model is constructed as a three-layer network structure. The input layer inputs key vectors, response vectors, and the associative relation between vectors. The memory layer stores input vectors incrementally to corresponding classes. The associative layer builds

associative relations between classes. The method can incrementally learn key vectors and response vectors; store and recall both static information and temporal sequence information; and recall information from incomplete or noise-polluted inputs. Using the GAM model, Shen and Hasegawa demonstrated promising results of pattern recognition experiments using binary data, real-value data, and temporal sequences.

1.2.1.5.1 GAM Architecture

The input layer accepts any data that is encoded as a sparse distributed vector. These input vectors are called *key* and *response* vectors. The input layer receives the key vectors and response vectors. *Response vectors* are the outputs of the *key vectors*. The memory layer classifies the vectors into separate classes based on the similarity of the vectors falling within a threshold limit. The similarity between two vectors is measured through a distance calculation using normalized ***Euclidean distance***. The memory layer stores the input vectors incrementally to the corresponding classes as it receives the input vectors. If the input vector does not belong to an existing class in the memory layer, the GAM builds a new subnetwork in the memory layer to represent the new class. The GAM sends the class labels of subnetworks in the memory layer to the associative layer, and the associative layer builds relationships between the class of the key vector (the key class) and the class of the response vector (the response class) by using arrow edges. One node exists in the associative layer corresponding to one subnetwork in the memory layer. The arrow edges connecting these nodes represent the associative relationships between the classes. The beginning of an arrow edge indicates the key class; and the end of the arrow edge indicates the corresponding response class. The associative layer builds associative

relationships among the classes. The GAM can store and recall binary or non-binary information, learn key vectors and response vectors incrementally, realize many-to-many associations with no predefined conditions, store and recall both static and temporal sequence information, and recall information from incomplete or noise-polluted inputs. Experiments using binary data, real-value data, and temporal sequences show that GAM is an efficient system.

GAM at first realizes auto-association, and then hetero-association as humans initially recognize or recall a class with a garbled or incomplete key vector, and then associate it with other classes according to the recognition of the key vector. A pattern recognition or pattern completion process uses auto-associative information and association between classes uses hetero-associative information.

1.2.1.5.2 GAM Analysis

The complete General Associative Memory (GAM) algorithm was analyzed as a baseline algorithm for this dissertation research. It was observed that the GAM algorithm has an advantage as its datapath can be designed using the SIMD concepts. Also this algorithm fits well for a hybrid system level design as the control logic of the algorithm can be designed in CMOS, while the datapath and memory operations can be designed with a nanotechnology.

Since the goal of this dissertation was to develop a methodology for hardware design, we realized that there is no need to design the complete GAM system. We rather identified one most common and critical component that is widely used in GAM and many other similar associative memory architectures. Thus Euclidean Distance Calculator was

identified for this methodology development work. Also, the reason the example of the Euclidean Distance calculator was used for this research is that it is widely applied by many Neural Network and similar algorithms in software. However, there is no hardware implementation available or even published. Moreover, for the application areas of pattern recognition, facial recognition, robot vision, Digital Signal Processing, voice recognition, big data analysis, and database mining, all of those algorithms require to process massively parallel large number of wide-word input vector/data and therefore, we need a hardware system that can handle those large number of wide input vectors or neurons efficiently. Conventional CMOS technology is not enough for handling any such massively parallel applications, and as a result, this dissertation proposes an alternate, memristor-based nanotechnology using stateful IMPLY based FPGA design, MsFPGA (Memristive stateful logic Field Programmable Gate Array).

This proposed MsFPGA is the new idea and development by itself, only motivated by the previous research on associative memories. It can be used for many other applications, the same way as CMOS-based FPGA architectures are being used now. However, to illustrate the proposed new device, we use the Euclidean Distance calculator, which can be applied as an important component in any of the above application areas listed. Besides, in this dissertation several potential applications of the proposed FPGA architecture and its associated design methodology are mentioned, such as pipelined and SIMD-like architectures, which are typical for neural network, machine learning, robot vision, and control related applications.

1.2.2 Part 2: Memristor Based Research

1.2.2.1 Development on Memristor Research

Memristive devices [1] are electrical resistance switches that can retain a state of internal resistance based on the history of applied voltage and current. These devices were theoretically conceived in the late 1960s and recent progress has led to fast, low-energy, high-endurance devices that can be scaled down to less than 10 nm and stacked in three dimensions [40]. Leon Chua in 1971 [1] published the theoretical description of memristors. Strukov et al. [2] in 2008 established the link between the memristor theory and experimental results for the first time. Snider et al. [81] showed that configurable crossbars are the easiest computational structures to fabricate at the nanoscale level and also to assemble them into larger structures. Likharev and Strukov in 2005 [6] predicted the development of hybrid CMOL integrated circuits that would extend Moore's Law to the few-nm range. They listed several important future applications for such circuits, including large-scale memories, reconfigurable digital circuits, and mixed-signal neuromorphic networks with high and promising performances in delay and power. Likharev [71] proposed a memristive nanowire crossbar array. Wei Lu et al. [32] in 2011 experimentally demonstrated 1 Kb hybrid CMOS/memristor passive crossbar memory. Govoreanu et al. [59] in 2011 demonstrated functioning memristive devices at the 10 nm scale. Yang et al. [83] recently published a paper where they reviewed the recent progress in the development of memristive devices with the performance requirements. Pierzchala and Perkowski [74] provided a crossbar architecture that is flexible for a programmable electronic hardware device or for an analog circuit whose input and output signals are analog or multi-valued in nature, and primarily continuous in time. Likharev et al. [89] proposed a technique for the spatial pattern recognition by implementing the Hopfield network

model using the CMOL crossbar network. Peng Li et al. [67] have developed memristor based design for memory and used it in a digital system that can be applied for spatial pattern recognition. Intel recently published [78] their research results using cross-bar neural network architecture based on memristor (phase change memory/PCM) synapses and spin neurons. Through estimates for common data processing applications this NN hardware shows 20X - 300X improvement in the computation energy when compared to the state of the art CMOS designs.

In a price/performance study for the Hierarchical Temporal Memory (HTM) model, Zaveri & Hammerstrom in 2011 [84] showed the comparative results of mixed-signal (MS) CMOL, digital CMOL, digital CMOS and Analog CMOS designs. At present, if manufacturing cost is considered as a measure of price, then obviously digital CMOS design would be the cheapest. But in the near future, as the CMOL technology matures, CMOL based designs seem to be more suitable for implementing computation and memory intensive applications (Zaveri & Hammerstrom, 2010) [85]. Also, at present, power is becoming the major measure of price, with silicon area being important, but in the second place of importance to power. Neuromorphic circuits may provide solutions to these latest design issues.

1.2.2.2 Summary on the Dissertation Research

Memristors are small devices that are able to hold a state, and therefore, this research proposes to take advantages of the physical characteristics of memristors for certain design, where data processing can be difficult or area-consuming with pure CMOS implementation [22][31]. Memristive computing is expected to be advantageous in large-scale, massively parallel architectures. As such, memristor-based design is explored for such applications in order to evaluate the possible performance gain on the circuit design aspects, such as,

the processing time, area and the power consumption compared to the conventional CMOS implementation.

The concepts and methodologies of this work were formulated based on the fact that memristors themselves can perform logical operations [3][4]. In pure CMOS implementations the datapath, memory and controllers must share die area. In contrast to this, in this research for the memristor nanowire crossbar implementation, the memristors are located above the CMOS layer in a physical layout. Because the memristor array is proposed to be fabricated on top of CMOS [6][8], this frees up CMOS die area, and thus reduces the total area of the combined CMOS and memristor circuits. Thus it enables more datapath and memory logic [3][4][21]. The CMOL-like architecture and methodology proposed in this dissertation are different from "Strukov and Likharev" proposed CMOL (CMOL is defined as hybrid CMOS/MOLecular device at nanowire intersections) [6][8][32][86]. In this proposed methodology, the pulses are generated in a memristor-CMOS hybrid signal generator and are controlling the datapath and memories built with memristor-Implication logic in CMOL-like nanowire crossbars. In contrast to "Strukov and Likharev" proposed CMOL, in this proposed MsCMOL (Memristive stateful CMOL) methodology, two memristors need to be addressed at a time to perform one logical transfer, while in their CMOL, only a single memristor is addressed at a time, which leads to less design flexibility. Also, the conceptual realization of the proposed memristor-IMPLY based circuits is based on the realization of combinational logic in space and time in such a way that a single memristor holds its state and is reused in many virtual IMPLY gates.

A major portion of the dissertation is devoted to the realization of combinational and sequential, pipelined logic in the proposed MsFPGA (Memristive stateful logic Field Programmable Gate array). This dissertation work did not concentrate on programming of memristor-based memories and connections, because these are well-known topics, known from the literature. The main contributions of this dissertation are the concepts and methodologies to design regular pipelined and SIMD-like architectures with stateful IMPLY memristive logic. The experimental results show substantial advantages of this new concept as compared to the classical CMOS FPGA in terms of area, power, and speed.

1.3 **Thesis Organization**

Chapter 1 discusses about the research motivation. Finding a hardware methodology for the biologically inspired associative memory models for the application areas of big data analysis, pattern recognition, robot motion, neural network etc. was the original motivation of this work. Therefore, the focus of this research was to provide a hardware methodology that is suitable for massively parallel and pipelined architecture and can be implemented with nanoscale technology. Memristor is a promising new technology and as such, this dissertation proposes a methodology of designing a massively parallel reconfigurable architecture using the IMPLY-memristor based nanowire crossbar.

Chapter 2 discusses about the fundamental theories of memristors and CMOL crossbar from the literature, how to design a stateful IMPLY gate with memristors, logic synthesis with IMPLY gate using proposed space-time based notation together with pulse

generators, proposed optimized design of critical logic gates, like XOR and a 1-bit Full adder with sneak-path protection, innovative concepts of 8-bit iterative adder design in a new type of 8x8 nanowire crossbar, sneak-path reduction and pipelining using the array of 8x8 nanowire crossbar blocks.

Chapter 3 introduces the proposed reconfigurable Hybrid memristor-CMOS MsFPGA (Memristive stateful logic Field Programmable Gate Array) design. The proposed architectural concepts behind the MsFPGA design such as: MsFSMD (Memristive stateful Finite State Machine with Datapath), Pulse Generator, use of MsRAM (Memristive stateful RAM), and block placement architecture are also discussed. The proposed pipelined architecture of MsFPGA with SIMD-like massive parallelism is presented as well.

Chapter 4 presents the concept of designing the Euclidean Distance Calculator as an innovative pipelined datatpath. For future comparison against the proposed MsFPGA, this chapter also presents the detailed implementation of such a datapath as a CMOS FPGA design using the Xilinx Vivado 2015.2 tool. The complete design was coded using the hardware description language VHDL, synthesized in Xilinx Vivado 2015.2 and analyzed for area, power, and delay.

Chapter 5 discusses the circuit implementation challenges of MsFPGA. The proposed MsCMOL, use of data MsRAM, array of 8x8 nanowire crossbar blocks, proposed sneak-path protection, proposed row-to-row data transfer, proposed 8-bit iterative adder

design were discussed. MsFPGA was compared with other published memristive FPGAs, for example, mrFPGA. Proposed MsFPGA is a reconfigurable system that can be designed with pipelined datapaths and massive parallelism. This parallelism can be designed by driving many such pipelines with one controller simultaneously, using the SIMD (Single Instruction Multiple Data)-like concept. Relation of this dissertation work to the recently published relevant paper on NVM is also discussed.

Chapter 6 introduces an innovative novel sneak-path protected IMPLY-memristive-nanowire crossbar circuit design methodology. For this purpose, an example of 8-bit Full iterative adder design was presented in detail. Also possible power consumption issues are discussed. Current literature was studied and compared.

Chapter 7 discusses about the performance study of proposed MsFPGA in detail. PSPICE simulations were performed on nanowire crossbars for the nanowire RC delay measurement. Besides, memristor-nanowire power and area estimation were presented in this chapter. Other benefits of memristors e.g., functioning as a pipeline was discussed.

Chapter 8 presents the results and comparative analysis of CMOS FPGA versus the proposed memristive FPGA, MsFPGA for the Euclidean Distance pipeline.

Chapter 9 is a short chapter on the conclusions and contributions of this dissertation research.

2 MEMRISTOR

The existence of the memristor, the fourth fundamental passive circuit element, was theoretically predicted in 1971 by Leon O. Chua [1], but not experimentally validated until 2008 by HP Labs [2][33]. A memristor is essentially a nonvolatile nanoscale programmable resistor, memory resistor — whose resistance, or memristance, is changed by applying a voltage across the device. Chua [1] defined memristor as shown in Figure 1 (symbol shown in Figure 2) as a previously missing relation between the flux, φ and the charge, q and therefore yielding the defining relation, $M(\boldsymbol{q}) = \frac{\boldsymbol{d\varphi}}{\boldsymbol{dq}}$.

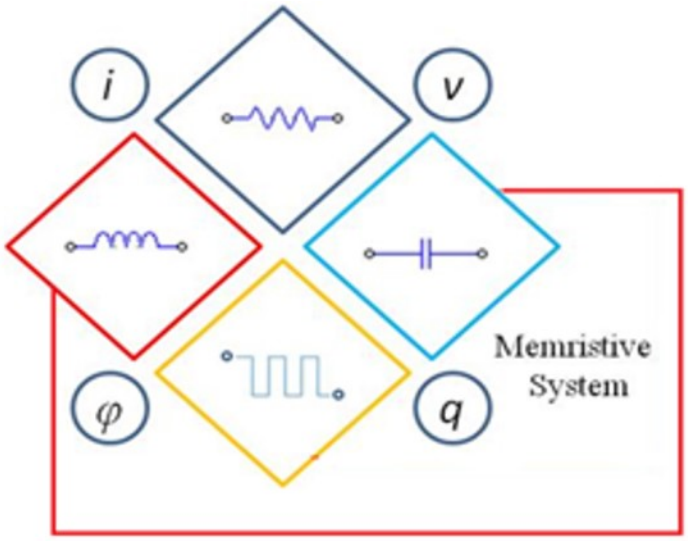


Figure 1: Fundamental Passive Elements [1].

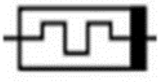


Figure 2: Symbol of Memristor [3].

As shown in Figure 3, the I-V curves of memristors form pinched hysteresis loops, and the forms of these loops depend on the amplitudes and frequencies of the input voltage signals.

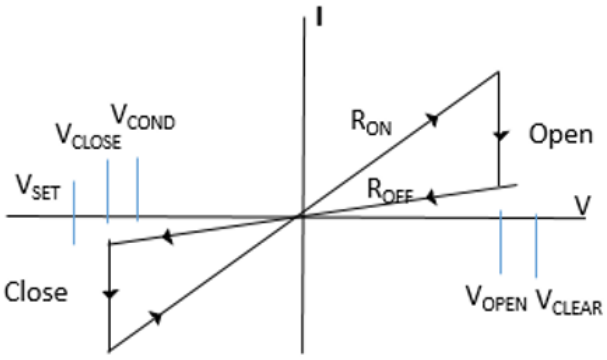
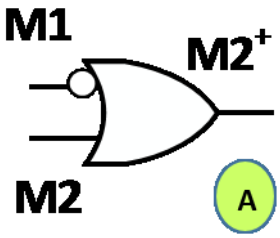


Figure 3: Memristor I-V curve shows hysteresis loop [3].

This phenomenon defines a state variable, which determines the memristor's instantaneous resistance also known as the memristance.

2.1 **Stateful IMPLY memristor**

Memristors can be binary, meaning that they can have two distinct states of resistivity. Kuekes [3][4] in 2008 showed that the material implication logic operation can be efficiently implemented using memristors. Binary memristors can be used to perform stateful logic [3][4][5][7][11][12][94] which allows for direct implementation of logical computations within memristive crossbars (nano crossbars with memristors at intersections), assuming that one first develops some methodologies for synthesis of digital systems with combinational and sequential datapaths built from working memristors that are pulsed from the memristor-CMOS hybrid control block.



Case	M1	M2	$M2^+$
1	0	0	1
2	0	1	1
3	1	1	1
4	1	0	0

B

Figure 4: A. Implication (IMPLY) Logic Gate B. Truth Table [3].

The circuit in Figure 4-A shows the realization of the implication gate with two memristors, which enables stateful memristor logic [3]. The truth table of this circuit is presented in Figure 4-B. Here the memristors are assumed to be bistable linear devices having the on-resistance R_{ON} and the off-resistance R_{OFF}, where the resistance ratio is assumed to satisfy $R_{OFF}/R_{ON} \gg 1$. The series resistance R_G is chosen as $R_{ON} < R_G < R_{OFF}$.

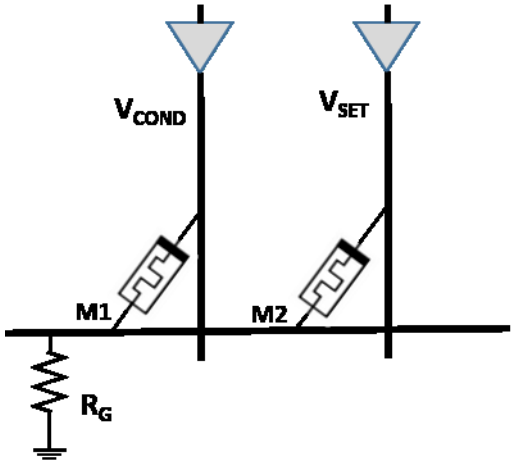


Figure 5: Implication (IMPLY) Logic: Realization with Two Memristors [3].

In Figure 5, Memristor M1 is driven with a conditional voltage V_{COND} and memristor M2 is driven with a voltage V_{SET}, where the next state of M2 depends on the IMPLY stateful

logic operation ($|V_{SET}| > |V_{COND}|$). Memristor M2 is called the "working memristor" (WM) and memristor M1 is called the "input memristor" (IM). The flow of current through the M2 memristor changes its memristance and consequently changes the memorized internal state. This is how the input memristor M1 state affects the output state $M2^{+}$ of memristor M2. Memristors can be reset to state '0' by applying a V_{CLEAR} voltage. Memristors hold a logical state as a resistance value and not as a voltage value. Resistance values R_{OFF} and R_{ON} represent logical state '0' and logical state '1', respectively.

2.2 Logic Synthesis with memristors

A hybrid reconfigurable system level design with memristors and CMOS is proposed in this dissertation. A space-time based circuit notation is used in the methodology proposed in this section for the memristor based logic design based on the implication logic gate. The implication logic gate is used for this work because it is a universal logic component that can implement any function and also memristor implementations are very efficient using this logic gate. The notion of the memristor based *stateful* logic gate is that the output of the gate represents the next state of one of the inputs to the gate. Since a memristor can hold a state, either as a '0' or as a '1', the output of a memristive logic gate will hold the next state of the input as a logic ***high*** or as a logic ***low***.

In a space-time based notation, s*pace* represents the design area and *time* represents the sequential changes of states related to delay or speed of the design [102]. The symbolic drawing of the space-time based notation to represent a memristive implication gate is shown in Figure 6.

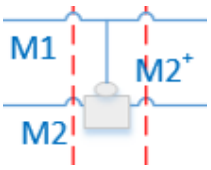


Figure 6: Symbol for space-time notation [102].

In this drawing, each horizontal line represents a memristive nanowire. For the proposed methodology, two vertical nanowires need to be selected simultaneously to realize a single logical operation (IMPLY operation). In the symbolic drawing, horizontal lines represent physical memristors as they change value in time and vertical lines represent moments (time) of transfer between two memristors which realizes the IMPLY operation. This way, the timing pulses can be illustrated to move data sequentially from the left to the right as shown in Figure 7.

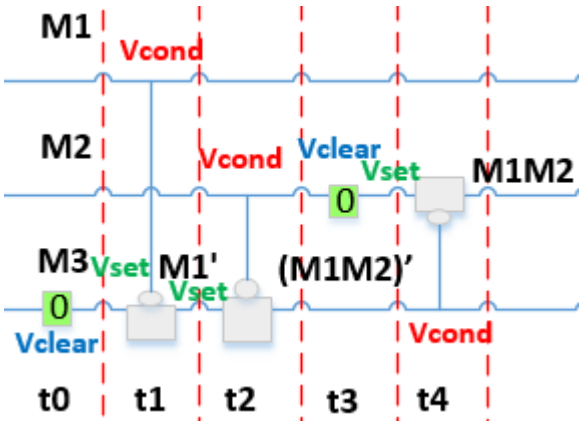


*Figure 7: Implementation of NAND/AND Gate using space-time notation. "Reset" operation on memristors using the VCLEAR voltage is indicated by a '0'. At time t0, signal VCLEAR is presented to working memristor M3 which initialized its value to '0'. At t1, signal V_{COND} is presented to WM M1 and signal V_{SET} to WM M3, which causes the state of M3 to be M1'+0=M1'. At t2, the state of M3 becomes M1'+M2'=(M1*M2)' (DeMorgan's Law). At t3, the WM M2 is scheduled to be reused by clearing it. Finally, at t4, the negated value of M3 is added to M2. Thus $M2^+ = 0+((M1M2)')' = (M1M2)$. $M2^+$ is the next state of M2. [102]*

The red dotted lines on both sides of each gate denote one timing pulse. The two memristors on which IMPLY operation (M1 → M3) is executed are connected with a vertical line and receive pulses of V_{COND} and V_{SET} voltages simultaneously. The source memristor M1 receives V_{COND} and the target memristor M3 receives V_{SET} voltages. Also V_{CLEAR} voltage is applied to reset any memristor to the logic '0' state (e.g. state '0' is shown in time-step t0, t3 in Figure 7). It is obvious that IMPLY and '0' operators make a universal logic system, because a NAND gate can be created from two IMPLY gates: $B \rightarrow (A \rightarrow 0) = (A' + 0) + B' = (AB)'$. It is well-known that NAND is a universal gate. The design process is illustrated in Figure 7 with a NAND/AND gate implementation [102].

A single CMOS clock drives the proposed entire hybrid system. The CMOS CLK drives the counter, which generates the sequences for the Memristive stateful RAM (MsRAM) and consequently the CLK or micro-pulse for each implication logical operation in the memristive crossbar datapath. In this methodology, '**t**' represents the clock cycle and the time steps are denoted as t0, t1, t2, t3, …, etc. starting with the reset operation at time-step t0 and IMPLY logical operation starting at time-step t1 as shown in Figure 7.

Using the design methodology introduced in this dissertation, any combinational or sequential circuit can be designed and optimized for area and delay. The design of an XOR gate, a useful gate in many applications, and in particular in the applications of the MsRAM in the proposed pulse generator, is presented next. Proposed XOR design uses only two working memristors and seven pulses which makes this design competitive in larger logic circuits. Figure 8 is an important figure, because problems that appear in logic synthesis of larger memristive circuits in stateful logic can be illustrated through this design [102].

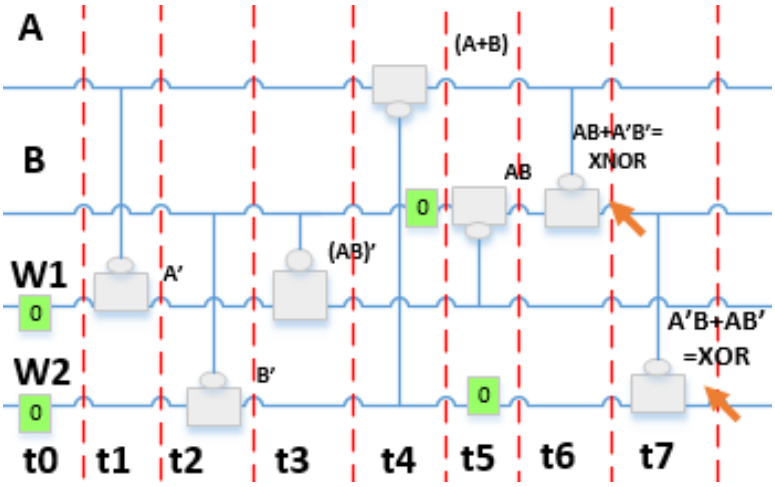


Figure 8: Space-time notation of an Implication based circuit for a 2-Input XOR/XNOR Gate [102].

Design of a two-input, A & B, XOR gate is presented in Figure 8 and Table 2-1, which shows that it requires only 7 micro-pulses and 4 memristors to realize this gate. Therefore, the minimum design requirements for an *n*-input XOR gate is set as (7*(*n*-1) + 1) pulses and *n* working memristors (W_1, W_2, .. W_n) if all memristors are reused to realize IMPLY gates [102].

In Figure 8, in moment **t4** we simultaneously execute transfer from W2 to A (A′ = W2′ + A) and we clear memristor B. Thus three control signals are created simultaneously. V_{CLEAR} for memristor B, V_{SET} for memristor A and V_{COND} for memristor W2. This is possible because in this proposed architecture every column can be addressed individually, in contrast to the classical memristor crossbar.

Table 2-1: XOR GATE IMPLEMENTATION IN SPACE-TIME NOTATION [102].

Timing pulse	Implication Gate Structure	Logical Operation	Comments
t0	PI: A, B WM: W1, W2		W1 reset to state0 W2 reset to state0
t1	$A \rightarrow 0$	A'	Inverter
t2	$B \rightarrow 0$	B'	Inverter
t3	$B \rightarrow A'$	$A' + B' = (A.B)'$	NAND Gate
t4	$B' \rightarrow A$ Memristor B	$A + B$	OR Gate B reset to state0
t5	$(AB)' \rightarrow 0$ Memristor W2	$A.B$	AND Gate W2 reset to state0
t6	$(A + B) \rightarrow AB$	$AB + A'B'$	XNOR Gate
t7	$(A'B' + AB) \rightarrow 0$	$AB' + A'B$	XOR Gate

A 1-bit full adder is fundamental to the proposed Euclidean Distance (ED) pipeline design. An innovative adder is designed from stateful memristive IMPLY gates with five working memristors and only 18 micro-pulses to generate the sum and carry signals as presented in Figure 9. This adder is designed with a much improved sneak-path current protection compared to other published adder design [5]. An 8-bit adder circuit was implemented in an 8x8 crossbar nanowire, in which we reset the three primary memristors and four working memristors of each adder bit (located in each row of the 8x8 crossbar) after completing the logical operations. Therefore, this design provides much improved sneak-path current protection. The only memristor that holds the "sum" bit cannot be cleared, but this memristor is unable to contribute to sneak-path current as there is no direct sink path to Gnd. Similarly, an 8-bit adder requires 165 micro-pulses for the digital design illustrated in the symbolic space-time notation [102]. While Figure 9 presents a single bit adder in symbolic notation, Figure 10 presents the corresponding 8-bit adder where every

row realizes a single bit adder [102]. This notation from Figure 10 is not symbolic and extends the physical circuit from Figure 5.

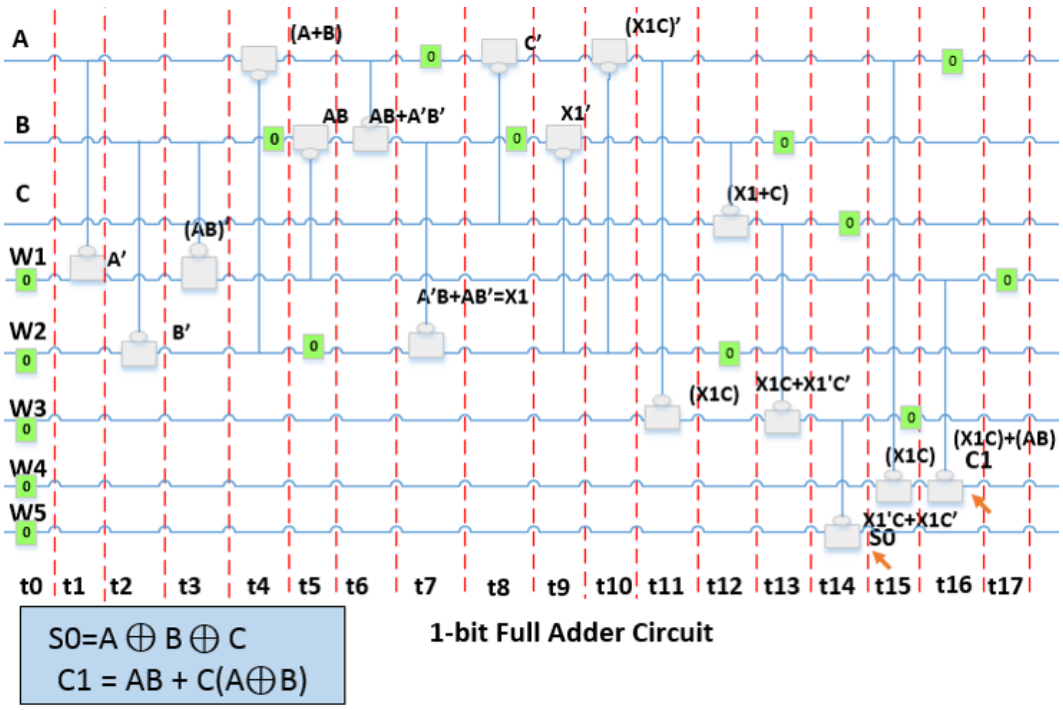


Figure 9: Space-time notation for 1-bit Full Adder circuit with sneak-path protection [102].

The presented adder circuit is designed carefully to avoid sneak-path currents. When carry and sum for a given bit are calculated, the working memristors that are no longer needed are cleared by setting them to logic state '0'. The remaining one memristor located in column 8 of each row of the 8×8 nanowire crossbar from Figure 10 holds the sum bit of each bit adder.

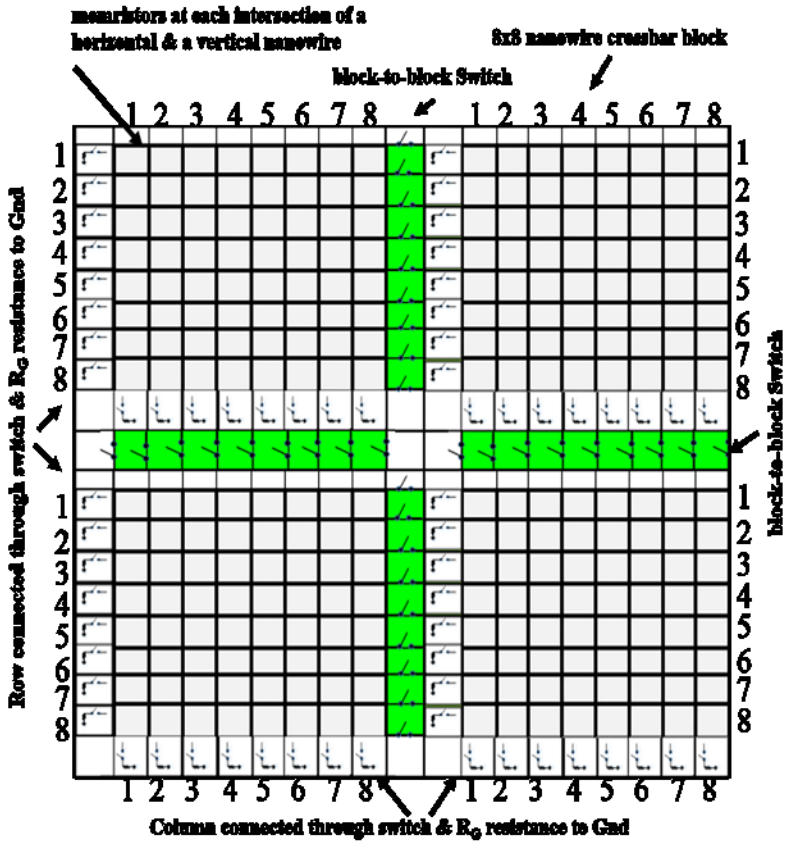


Figure 10: An example architecture of four 8×8 nanowire crossbar blocks. Block to block horizontal and vertical connections are made through switches in nanowire layer. Each horizontal and vertical nanowire is connected to Ground through switch and load resistance R_G to provide protection from sneak-path current. CMOS decoders are placed beneath the nanowire crossbar layer in a physical layout [102].

In general, using the proposed space-time based circuit design method, the designer should utilize the minimum number of memristors in the crossbar to optimize the die area. One of the techniques for saving area is to reuse the memristor by resetting the previously used working memristors to logic '0' state using the V_{CLEAR} voltage, as illustrated earlier. During this reset step, this same memristor cannot participate in a logical operation, but it can be reused for a logical operation at any later time-step. Another technique to optimize the timing of the design is to reset multiple nanowires to the logic '0' state in a single

micro-pulse. Also, during this reset operation, simultaneously, a logical operation between two other memristors can take place. This way the total delay of the design can be reduced.

In conventional CMOS circuit design there is a distinction between the combinational logic stage and the sequential logic stage as sequential components are driven by a CLK signal in a synchronized design, whereas, in IMPLY-memristor based design, no such distinct separation exists. For example, in Figure 11, three DFFs are serially connected and the output of the third FF is connected to the input of the first FF through an AND gate. This circuit is realized in Figure 12 with IMPLY-memristor based space-time design technique and shows that there is no distinct separation to draw the complete circuit presented in Figure 11. This is an obvious advantage for the proposed IMPLY-memristor based logic design using the space-time technique.

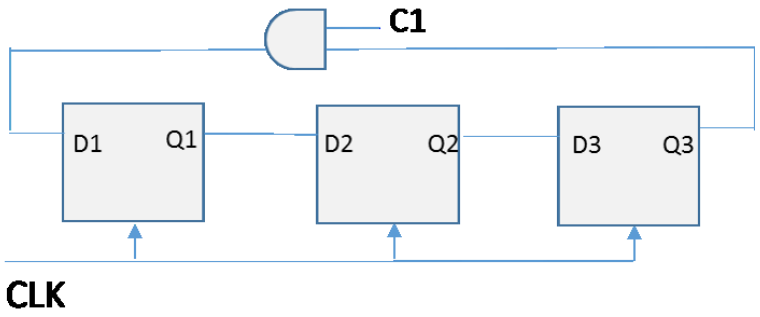


Figure 11: Combinational and Sequential components in CMOS Implementation.

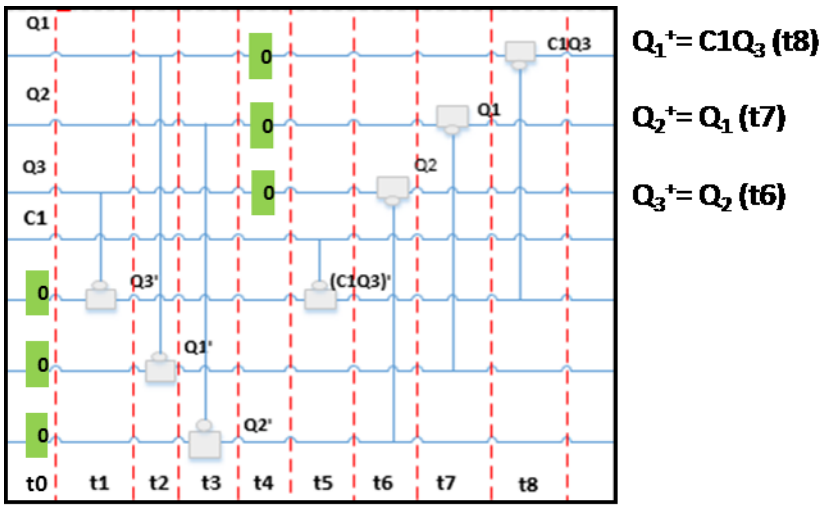


Figure 12: No Separation between Combinatorial and Sequential Logic in IMPLY-memristor Design.

2.3 CMOL Crossbar

One of the main driving forces of memristor technology is the prospect of crossbar architectures with very large numbers of memristive devices [8][32][86]. It is a 2D array, which consists of two perpendicular nanowire layers. The nanowires act as the top and bottom electrodes of memristors, and they can be patterned for example by e-beam lithography complemented with reactive ion etching, and lift-off processing for the upper nanowire layer [8][32][86]. A typical nanowire half pitch of the nanowires in the reported physical memristive crossbars, so far, is in the range of 30 nm−100 nm [20]. The memristive material is laid between the two nanowire layers, and as a result, a memristor is formed at each cross-point of two nanowires [8][32][86].

The “Strukov and Likharev” proposed CMOL memristor interfacing [8][94] is described below.

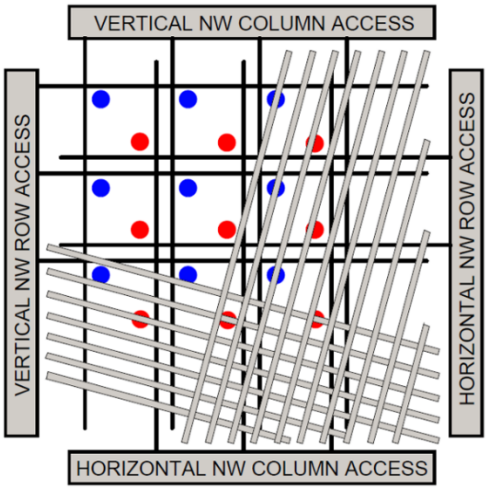


Figure 13: "Strukov and Likharev" CMOL Memristor Architecture [8][94].

In Figure 13 the CMOS cells are represented as square tiles, with circular interfaces to the nanowire crossbar. The nanowire crossbar is represented as a mesh of line segments. Each CMOS cell is addressed by four microwires that are connected to the address decoders represented at the perimeters of the circuit. The four-line addressing is used in order to independently control the horizontal and the vertical nanowires via the switches shown in the Figure 13. Although not shown here, there exists a memristor at each crossing of the nanowires. The CMOS pins are represented with red and blue circles. The red pins are connected to the lower layer or the crossbar, while the blue pins are connected to the upper layer.

The addressing of a memristive crossbar is established in a CMOL-type architecture by two sets of row and column decoders and pass transistors. The four-wire addressing allows the selection any pair of a horizontal and a vertical nanowire. This selection is then used for reading from and writing to the memristive crossbar.

The CMOL structure shown in Figure 14 was originally proposed by Likharev and Strukov [8], in which, only one memristor can be selected at a time using two nanowires in one CMOS cell. The nanowires are represented by yellow color and the memristor is located at the intersection of the two nanowires as shown by green dot.

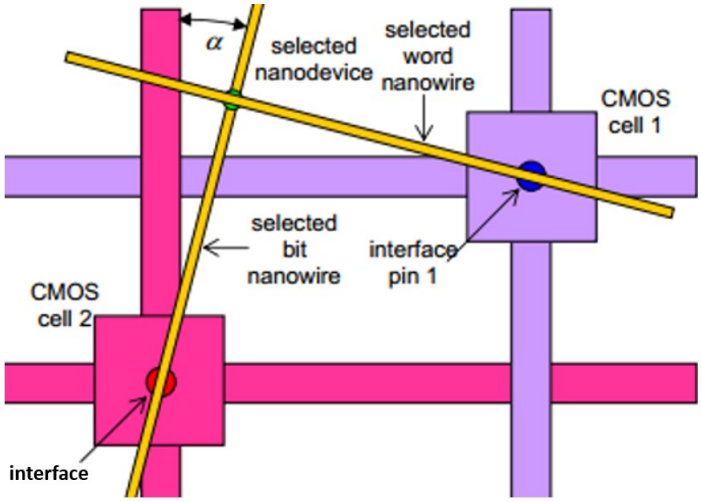


Figure 14: CMOL-memristor architecture by Likharev and Strukov [8].

A variation of the CMOL architecture called the Field Programmable Nanowire Interconnect (FPNI) was later proposed by Snider and Williams [45]. In FPNI, logic is performed by the CMOS layer, and only signal routing is realized by the memristive crossbar. Unlike Likharev and Strukov's CMOL-memristor architecture, more logical primitives than just inverters are needed at the CMOS layer. However, the Snider and Williams approach is also more like a classical FPGA-like circuit implementation and therefore is fundamentally different from the approach proposed in this dissertation. As

conventional CMOL only performs reconfiguration with memristors, while the logic circuits are designed with memristors in the proposed FPGA in this dissertation.

As mentioned, proposed CMOL-memristor architecture, MsCMOL is completely different from the conventional CMOL-memristor architecture [102]. A limitation of the conventional CMOL-memristor architecture is in the addressing scheme of the design where only a single horizontal and a single vertical nanowire can be selected at a time. This restriction to a single-junction selection may not be a problem in pure memory architectures as well as FPGA-like architectures based on configuring the space-realized memristor-based circuits, but in all stateful memristor logic applications it would be advantageous to be able to select multiple nanowires at once. This is because of the sequential nature of implementing logic operations in the realization of the stateful IMPLY gate. But even in this circuit two pulses are needed most of the time, one addressing the receiving memristor and another addressing the sending memristor. Thus the memory-like or FPGA-like (FPNI) addressing schemes and respective CMOLs are not applicable to this proposed design, as explained in a more detail later.

3 FPGA DESIGN USING MEMRISTORS

A Hybrid memristor-CMOS circuit can be used to implement reconfigurable Boolean logic circuits such as Field Programmable Gate Arrays (FPGAs) [6][8][9][10][13][30][86]. Since memristors act as non-volatile memories, memristive FPGA can retain its state when unpowered. Also, more than 90% of the area in classical FPGAs is consumed by the SRAM-based configuration bits [6][8][86], a memristive implementation can yield much higher logic gate density than is available in a pure CMOS implementation. Flash EEPROM based FPGAs are smaller and much less powerful and are not considered here.

3.1 Concepts Behind MsFPGA Architecture

Using memristor technology, we have the possibility of designing a massively parallel, programmable architecture with a very large number of memristive devices in a crossbar configuration [102]. Therefore, this dissertation work proposes a system-level architecture that significantly benefits from the small size characteristics of the memristive devices and nanowires. Moreover, the proposed design is capable of handling very wide word input vector lengths, and processing a large number of vectors (thanks to pipelined and SIMD-like architectures [95]). Multiple pipelines can be driven simultaneously by a single controller in the proposed ***MsFPGA (Memristive stateful logic Field Programmable Gate Array)*** [102]. The proposed MsFPGA allows the implementation of massively parallel computation by leveraging a massively parallel architecture with a very large number of

memristive devices in a crossbar as discussed in section 3.2.

The proposed hybrid memristor-CMOS reconfigurable system level architecture, MsFPGA (Memristive stateful logic Field Programmable Gate Array) is presented in Figure 15 [102].

3.1.1 Memristive stateful Finite State Machine with Datapath (MsFSMD)

A well-known model of a Finite State Machine with Datapath (FSMD) is a digital system composed of a finite-state machine controller, and a datapath. This model is used in centrally controlled pipelined and SIMD-like architectures. However, the FSMD concept cannot be applied to the proposed hybrid design with memristors. Therefore, a novel *non-von Neumann* architectural concept is proposed for the proposed MsFPGA design.

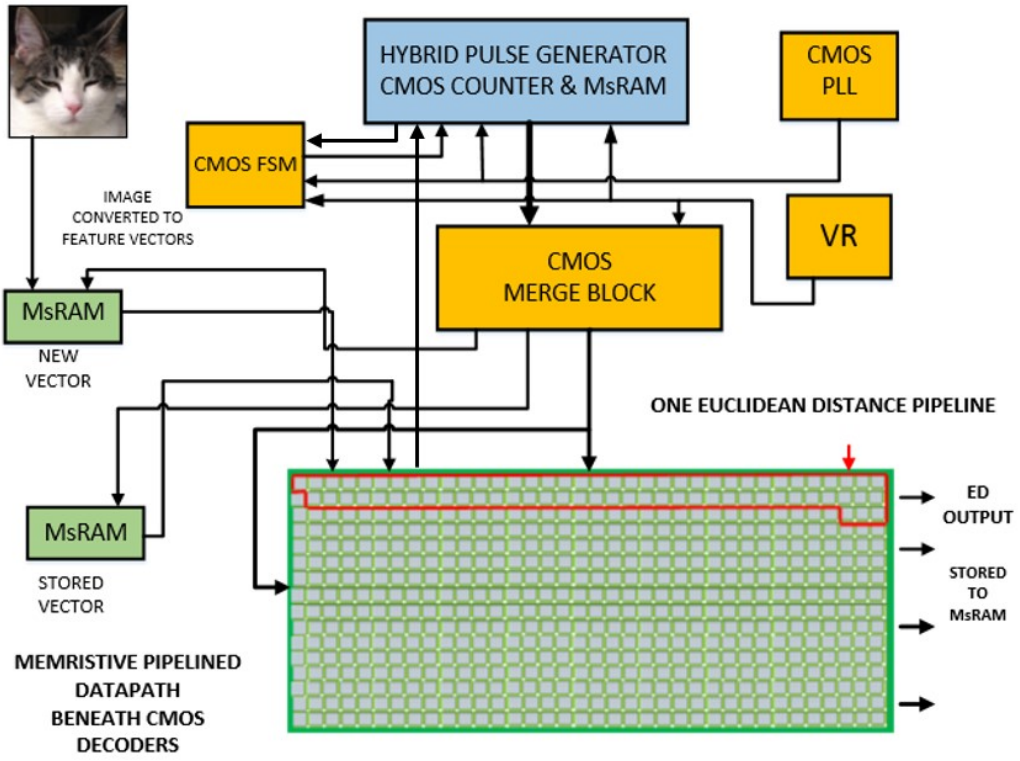


Figure 15: Proposed Memristive stateful logic Field Programmable Gate Array (MsFPGA). The details of the "Hybrid Pulse Generator" and the "CMOS Merge Block" are shown in Figure 17. The red polygon represents one pipeline of the proposed memristive ED architecture and the implementation is illustrated in Figure 41 in Chapter 7. Color code: Green- memristor nanowire crossbar, Yellow- CMOS, Blue- Hybrid circuitry [102].

The proposed methodology provides a novel general new architecture model, ***Memristive stateful Finite State Machine with Datapath (MsFSMD)*** as shown in Figure 16 [102]. Like conventional *FSMD*, this proposed system is also a digital system that includes a *finite-state machine*, and a *datapath*, but all logic is implemented with memristors, which changes timing and design methods used. Besides, the MsFSMD model has an additional control block called the ***pulse generator*** [102]. The pulse generator can be defined as the brain of the proposed MsFPGA. The pulse generation block contains the ***Memristive stateful RAM (MsRAM)*** and a CMOS counter [102]. The usage of the

MsRAM is another innovation of this dissertation work. This MsRAM contains all the configuration information required to realize the virtual logic circuit in the memristive nanowire crossbar datapath [102]. Unlike traditional von Neumann architecture, the proposed MsFSMD datapath can contain both datapath and memory without any distinct separation between them.

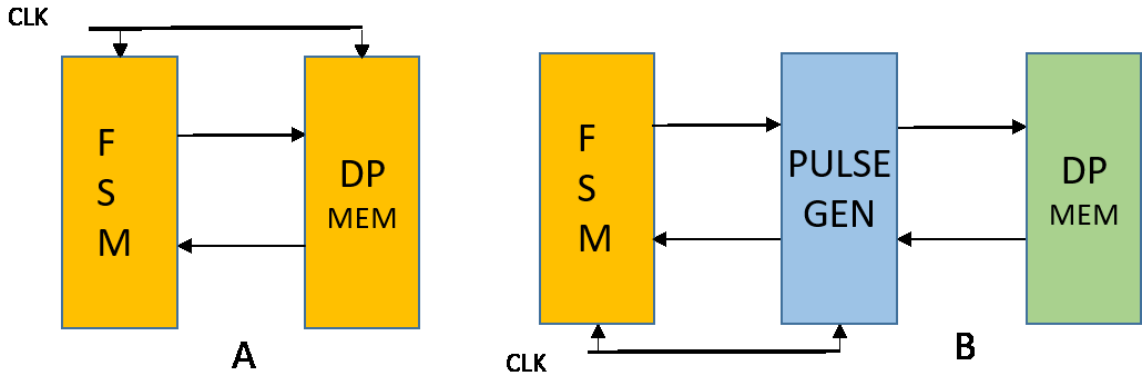


Figure 16: A. Conventional FSMD (Finite State Machine with Datapath) B. MsFSMD (Memristive stateful Finite State Machine with Datapath). The Pulse Generator block contains a CMOS counter and a Memristive stateful MsRAM. Color code: Yellow-CMOS, Blue-Hybrid CMOS-memristor, Green-Memristor nanowire crossbar.

In a conventional FSMD, the FSM controls the register-transfer operations in the datapath. The datapath performs data processing operations and sends the flags ('yes' or 'no') back to the FSM as shown in Figure 16. The proposed MsFSMD has two innovative properties:

1. Like FSMD, this proposed system is also a digital system that includes a finite-state machine, and Datapath, but all logic is implemented with memristors, which changes timing and design methods used.

2. The MsFSMD model has an additional control block called the ***pulse generator*** as shown in Figure 17 and presented in section 3.1.2.

3.1.2 Pulse Generator

The pulse generator can be defined as the brain of the proposed MsFPGA. The pulse generation block consists of a CMOS counter and a Memristive stateful RAM (MsRAM). This MsRAM contains all the configuration information required to realize the virtual logic circuit in the datapath. For instance, the datapath can have various arithmetic components such as an adder, subtractor, square, square-root, comparator, divider, and multiplier. Although the information in MsRAM is stored as memristances, the output values of MsRAM are available as voltages. The Voltage Regulator (VR) creates voltages $V_{SET,}$ V_{COND} and V_{CLEAR}. These voltages are fed to the merge block in which many multiplexers controlled from the Pulse Generation block select the controlling voltages for each column in the datapath blocks. The use of voltages in the realization of stateful IMPLY operations was already explained in Figure 7.

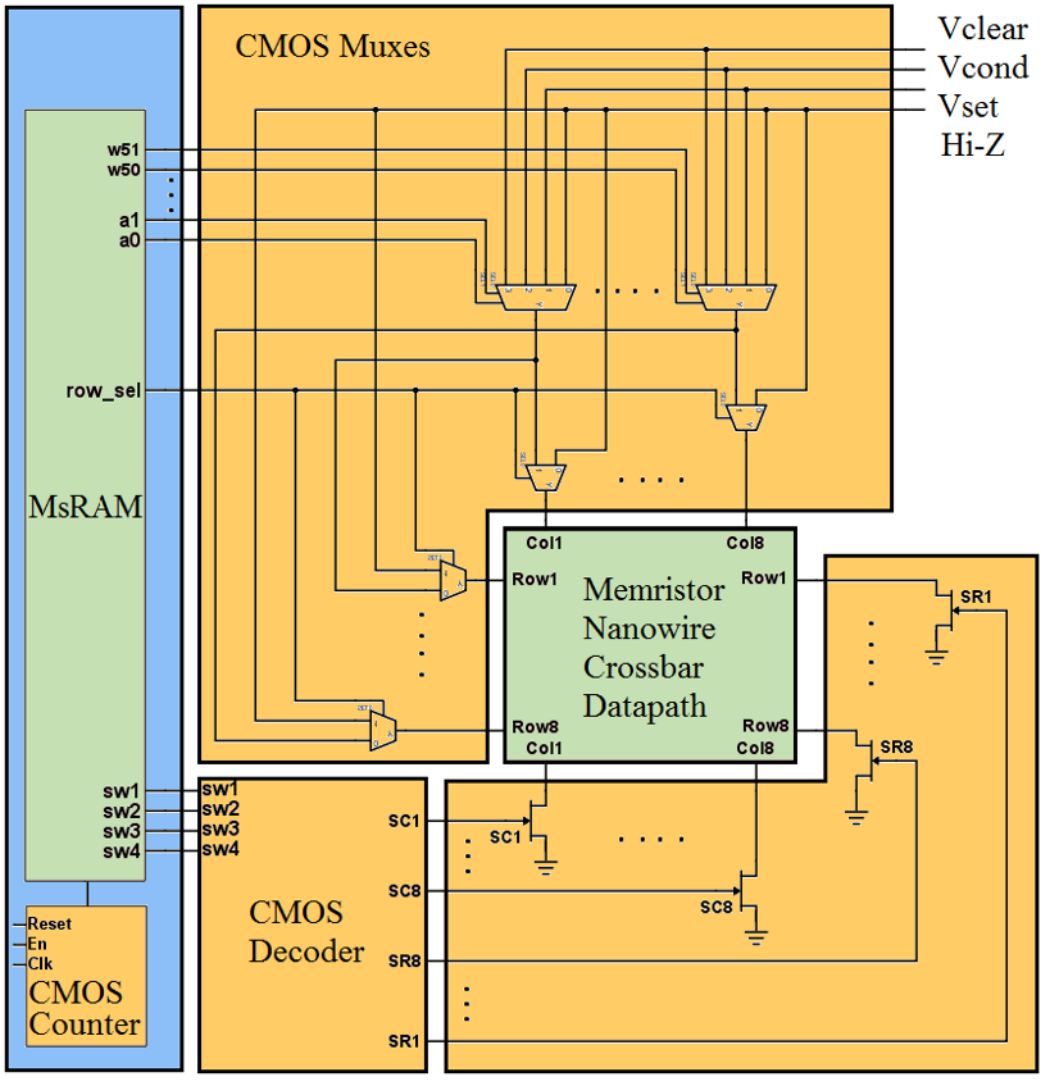


Figure 17: Proposed controller for the CMOS-memristor hybrid design (Pulse Generator with Merge Block). Color code: Green-memristor nanowire crossbar, Yellow- CMOS, Blue- hybrid circuitry [102].

Figure 17 explains how the proposed architecture switches between row nanowires and column nanowires [102]. While the control voltages in Figure 5 are only presented to the column nanowires, and the row nanowires are connected to Gnd, the proposed architecture from Figure 17 allows switching the roles of rows and columns. This is done

using the CMOS multiplexers. Therefore, it is possible that the horizontal nanowires are given control voltages while the vertical nanowires are connected to Gnd.

The proposed architecture then includes:

1. voltage multiplexers,

2. row select multiplexers,

3. column select multiplexers and

4. 4-to-16 one-hot decoder for switch to ground selection.

All these components are realized in CMOS. Buffers can be inserted to prevent signal attenuation in case of long lines.

3.1.3 Memristive stateful RAM (MsRAM)

An MsRAM is made out of memristors. The control data in MsRAM, located in the *Pulse Generator* are described with the encoding table to generate the pulses for the datapath as shown in Figure 18 [102]. This table illustrates controls for a portion of 8-bit Full-adder circuit as per Figure 9 in Chapter 2. It can be compared to the well-known tables that illustrate contents of ROMs in classical designs. Rows correspond to addresses given in time by a controlling CMOS counter and to the time pulses given to the datapath by the pulse generator. The right-most five columns are kept for row/column selection and providing sneak-path protection. A single bit row_sel signal is used to distinguish whether the voltages are applied to a row or to a column. When the row_sel signal is '1' the voltages are applied (through columns) to a row, and when the row_sel signal is '0' the voltages are applied (through rows) to a column. Therefore, when the row_sel signal is '1', voltages are

applied to the row (through columns) from the pre-programmed MsRAM, and the column select will be selecting high impedance state (Hi-Z). Also, a 4-bit CMOS one-hot decoder (Figure 17 shows columns sw1, sw2, sw3, sw4) is used to select one wire (1 row or 1 column) from a total of sixteen wires (8 rows and 8 columns) to connect to the ground at a time. This feature ensures that current can sink through only one wire to Gnd, and thus providing protection from sneak-path current.

The MsRAM controls the two select lines of a mux (control signals on the output of pulse generator). Data inputs to this mux are the voltages: for controls 00 – HiZ (High Impedance State), 10 - V_{COND}, 01 - V_{SET} and 11 - V_{CLEAR} that are selected to control the vertical/horizontal nanowires to perform the stateful logical operations in the memristor crossbar datapath. Since the proposed design provided protection from sneak-path current by adding few additional cycles in Figure 9, the MsRAM got extended with the decoder control bits as presented in Figure 18 [102].

time	Logical Operation	output	a0a1	b0b1	c0c1	w10w11	w20w21	w30w31	w40w41	w50w51	row_sel	sw4	sw3	sw2	sw1
Bit 0															
t0	A0,B0,C0 from MsRAM	copy	A0	B0	C0	00	00	00	00	00	1	0	0	0	1
t1	A->W1(clear)	A'	10	B0	C0	01	00	00	00	00	1	0	0	0	1
t2	B->W2(clear)	B'	00	10	C0	00	01	00	00	00	1	0	0	0	1
t3	B->W1	(AB)'	00	10	C0	01	00	00	00	00	1	0	0	0	1
t4	W2->A, clear->B	A+B	01	11	C0	00	10	00	00	00	1	0	0	0	1
t5	W1->B(clear),clear->W2	AB	00	01	C0	10	11	00	00	00	1	0	0	0	1
t6	A->B	AB+A'B'	10	01	C0	00	00	00	00	00	1	0	0	0	1
t7	B->W2(clear), clear->A	X1 = A'B+AB'	11	10	C0	00	01	00	00	00	1	0	0	0	1
t8	C->A(clear),clear->B	C'	01	11	10	00	00	00	00	00	1	0	0	0	1
t9	W2->B(clear)	X1'	00	01	00	00	10	00	00	00	1	0	0	0	1
t10	W2->A	(X1C)'	01	00	00	00	10	00	00	00	1	0	0	0	1
t11	A->W3(clear)	X1C	10	00	00	00	00	01	00	00	1	0	0	0	1
t12	B->C, clear->W2	X1+C	00	10	01	00	11	00	00	00	1	0	0	0	1
t13	C->W3, clear->B	X1C+X1'C'	00	11	10	00	00	01	00	00	1	0	0	0	1
t14	W3->W5(clear),clear->C	S0=X1'C+X1C'	00	00	11	00	00	10	00	01	1	0	0	0	1
t15	A->W4(clear),clear->W3	X1C	10	00	00	00	00	11	01	S0	1	0	0	0	1
t16	W1->W4,clear->A	C1= X1C+AB	11	00	00	10	00	00	01	S0	1	0	0	0	1
t17	clear->W1	clear	00	00	00	11	00	00	00	S0	1	0	0	0	1
t18	W4(r1)->W4(r2)	copy	10	01	00	00	00	00	00	00	0	0	0	0	0
t19	clear->W4(r1)	clear	11	00	00	00	00	00	00	00	0	0	0	0	0
t20	W4(r2)->C(r2)	copy	00	00	01	00	00	00	10	00	1	0	0	1	0
Bit 1															
t21	A1,B1 from MsRAM	copy	A1	B1	C1	00	00	00	00	00	1	0	0	1	0
t22	A->W1(clear)	A'	10	B1	C1	01	00	00	00	00	1	0	0	1	0
t23	B->W2(clear)	B'	00	10	C1	00	01	00	00	00	1	0	0	1	0
t24	B->W1	(AB)'	00	10	C1	01	00	00	00	00	1	0	0	1	0
t25	W2->A, clear->B	A+B	01	11	C1	00	10	00	00	00	1	0	0	1	0
t26	W1->B(clear),clear->W2	AB	00	01	C1	10	11	00	00	00	1	0	0	1	0
t27	A->B	AB+A'B'	10	01	C1	00	00	00	00	00	1	0	0	1	0
t28	B->W2(clear), clear->A	X1 = A'B+AB'	11	10	C1	00	01	00	00	00	1	0	0	1	0
t29	C->A(clear),clear->B	C'	01	11	10	00	00	00	00	00	1	0	0	1	0
t30	W2->B(clear)	X1'	00	01	00	00	10	00	00	00	1	0	0	1	0
t31	W2->A	(X1C)'	01	00	00	00	10	00	00	00	1	0	0	1	0
t32	A->W3(clear)	X1C	10	00	00	00	00	01	00	00	1	0	0	1	0
t33	B->C, clear->W2	X1+C	00	10	01	00	11	00	00	00	1	0	0	1	0
t34	C->W3, clear->B	X1C+X1'C'	00	11	10	00	00	01	00	00	1	0	0	1	0
t35	W3->W5(clear),clear->C	S1=X1'C+X1C'	00	00	11	00	00	10	00	01	1	0	0	1	0
t36	A->W4(clear),clear->W3	X1C	10	00	00	00	00	11	01	S1	1	0	0	1	0
t37	W1->W4,clear->A	C2= X1C+AB	11	00	00	10	00	00	01	S1	1	0	0	1	0
t38	clear->W1	clear	00	00	00	11	00	00	00	S1	1	0	0	1	0
t39	W4(r2)->W4(r3)	copy	00	10	01	00	00	00	00	00	0	0	0	0	0
t40	clear->W4(r1)	clear	00	11	00	00	00	00	00	00	0	0	0	0	0
t41	W4(r3)->C(r3)	copy	00	00	01	00	00	00	10	00	1	0	0	1	1
Bit 2															
t42	A2,B2 from MsRAM	copy	A2	B2	C2	00	00	00	00	00	1	0	0	1	1
t43	A->W1(clear)	A'	10	B2	C2	01	00	00	00	00	1	0	0	1	1

00	Hiz
10	V_{cond}
01	V_{set}
11	V_{clear}

Figure 18: Partial Encoding Table in MsRAM for an 8-bit iterative Full Adder realized in the Datapath (combination of control bits for various controlling voltages are: 00-HiZ, 01-VSET, 10-VCOND, 11-VCLEAR) [102].

3.1.4 Placement of Blocks and Connection Programming

The proposed Memristive stateful logic Field Programmable Gate Array (MsFPGA) architecture is completely reconfigurable and can be used for many applications, including those that require massive parallelism [102]. Massive parallelism in the example presented in this dissertation is based on pipelining and several pipelines operating in parallel. However, it should be obvious to the reader that advantages of proposed regular design are applicable also to Single Instruction Multiple Data (SIMD)-like architecture, systolic, and CMOL-like datapath-memory architectures which are typical to DSP, neural network and image processing. The proposed MsFPGA is particularly well-suited to regular designs with square or rectangular blocks executed in parallel or pipelined. Since the blocks (8x8 nanowire crossbar blocks) are placed in abutment, the horizontal connections are short and routing is simplified. This makes the architecture highly reconfigurable and also specifically suited for regular SIMD-like and pipelined architectures [102].

Since the example used in this dissertation does not have buses, and also the cells in the datapath are placed in abutment, connections are really simple. However, the proposed methodology is to send the output signals from a datapath block, located in the memristor-nanowire crossbar to the pulse generator block, in presence of buses, long connections or feedback connections in the datapath. The pulse generator is configured to send the control signals to the next datapath block located in the memristor-nanowire crossbar and also, the pulse-generator makes necessary communications with the CMOS FSM as shown in Figure 15.

3.2 Implementation of Proposed MsFPGA

3.2.1 Hybrid Architecture

As mentioned before, this dissertation proposes a memristive system-level architecture, MsFPGA, which is capable of handling large number of wide-word input vectors using the concepts of SIMD (Single Instruction Multiple Data) and pipelining [102]. The datapath, and any memory including the ***MsRAM (Memristive stateful RAM)*** of the pulse generation block of the proposed MsFPGA are designed in nanowire crossbar memristor-based technology, while the control logic- FSM controller, counter in pulse generator, multiplexers and decoder in the Merge block are designed in CMOS technology. Buffers can be inserted to prevent signal attenuation in case of long lines in the system.

3.2.2 Pipelined Architecture

In this dissertation, the general idea for MsFPGA design is illustrated with a specific example of a pipeline architecture for the Euclidean Distance (ED) calculation.

The straight-line distance between two points can be specified by using the Pythagorean formula. Euclidean Distance is the square root of the sum of the squares of the differences between corresponding values as shown in (1).

$$\mathrm{D(X,Y)} = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_n - y_n)^2}\,. \quad (1)$$

ED calculation is widely used in supervised and unsupervised learning, pattern recognition, and neural network algorithms for calculating distances between two neighboring neurons or vectors [19]. In addition to the above mentioned applications, ED is also used

[36][37][38][39][40][41][42][43][44][45] in Euclidean matrices, hierarchical clustering, phylogenetic analysis, molecular conformation in bioinformatics, dimensionality reduction in machine learning and statistics, natural language text processing, image processing, medical imaging, data mining, and big data analysis. Distance transformations use ED, and they are used in computer vision applications such as shape matching, pedestrian detection, human tracking, action recognition, robot motion planning or computation of morphological operations [39]. ED is also used in medial axis of a digital shape applied in surface reconstruction, shape simplification, volume representation and smoothing Voronoi diagrams applied in graphics and robot path planning. Several authors have discussed the growing importance of the ED calculation, which is incorporated in many important algorithms, and emphasized the need for fast ED calculation using a hardware realization.

The proposed ED pipeline hardware can be used as a powerful accelerator in any system, for both spatial and temporal pattern recognition [19]. For the proposed MsFPGA, two separate memories were used, one to hold any previously stored vectors and the other to store new incoming vectors as inputs to the ED pipeline [102]. The length of each vector can be ***'n'*** (where n is any integer), while each element of the vector is kept 8 bits wide in order to address image gray-scale values of 0-255. These two memories are also MsRAMs, but in contrast to the MsRAM used in the pulse generation block, they do not create output voltages, but the transfer from them to the datapath is done the same way as between memristor-based combinational blocks in the datapath which will be explained in Chapter 5 and Chapter 6 of this dissertation [102]. For simplification, circuits for the transfer of initial data to these memories are not considered in this dissertation.

3.2.3 Massively Parallel Architecture

As we see from the presented examples there is a very good match between the small size of memristors, the sequential realization of stateful IMPLY logic operations and a pipelined or Single Instruction, Multiple data (SIMD)-like design. Because of the sequential nature of combinational logic realization, the speed-up can be obtained only by implementing some form of parallelism. This is because:

1. Small size of memristor allows massively parallel architectures with low power dissipation.
2. Stateful design in which memristor stores a state, which allows "logic in memory" data-flow architectures, in particular pipelines, systolic and SIMD-like processors.
3. No flip-flops are necessary, which allows deep pipelines at small cost. Only pulses from the pulse generator are used.
4. The pulse generator is a relatively large circuit in several applications, but it can be shared among parallel blocks or pipelines.

Pipeline registers are not necessary because every memristor stores its value. This changes the proportion of cost going to combinational and sequential components and calls for massive pipelines, if possible. Due to the very small size of memristor crossbars, several pipelines can operate in parallel. The sizes of the blocks that are executed in parallel can be selected by the user, depending on the application. Similarly, non-pipelined SIMD-like designs can be realized making use of the small size of memristors combined with sequential realization of logic and relatively regular routing between blocks. Therefore, in

this dissertation a massively parallel architecture is proposed for system level design by both pipelining in the datapath and the ability to have many identical pipelines that are operating on separate data elements, where a single controller can drive many datapath blocks operating in parallel. It was assumed that the controlling machine is a CMOS-realized FSM that sends control signals to the hybrid Pulse Generator, which, in turn, controls many memristor-based pipelined datapaths. This parallelism along with the natural pipelining is expected to speed-up the overall design and to compensate for any delays due to the sequential nature of operations inside the 8×8 nanowire crossbars in the datapath that correspond to cells in standard FPGAs.

In this proposed methodology, a single controller can control multiple pipelines. The memristive controller is located in the proposed pulse generator (PG) and drives data in multiple pipelines with the same control. The proposed MsCMOL based datapath has memristors as well as CMOS. It is important to mention that when going to parallel pipelines all using a single controller, the controller increases only slightly with the big increase of the number of datatpaths. However, the ratio of memristors to CMOS in the datapath remains the same.

3.3 **Benefits brought by proposed architecture methodologies**

Proposed MsFSMD model is a good approximation to SIMD-like architectures in which there is one central controller (in case of this dissertation, FSM and Pulse Generator) and massively parallel regular array of relatively simple processors that communicate by abutting. All processors execute the same simple algorithm based on control signals from

the controller. These architectures include Cellular Automata, Image Processors based on Convolution, Morphological Processors, Sorters, Neural Networks and other.

Proposed pipeline architecture that is a good match with sequential nature of stateful memristor logic and with a regular style of designing circuits (also proposed in this dissertation work) based on abutting of hybrid blocks that are both pipelined and work in parallel. This dissertation compares this proposed design style with a classical CMOS FPGA. Therefore, all of the results from tables presented in this dissertation and related texts can be used to evaluate the proposed methodology relative to FPGA technology.

4 CMOS FPGA IMPLEMENTATION

To illustrate the concepts of the proposed methodology for this dissertation work, a Euclidean Distance (ED) processor was designed. The straight-line distance between two points in a multi-dimensional space can be specified by using the Pythagorean formula. ED is the square root of the sum of the squares of the differences between corresponding values as shown in (1).

$$\mathrm{D(X,Y)} = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_n - y_n)^2}. \quad (1)$$

The reason that the example of the Euclidean Distance calculator was chosen for this work is that it is widely used by many Neural Network and similar algorithms in software, but there is no hardware implementation available. Moreover, for the application areas of pattern recognition, facial recognition, robot vision, Digital Signal Processing, voice recognition, and big database analysis, the algorithms typically require lots of data and the processing of that data can be done by massively parallel data processing pipelines [102].

The example of Euclidean Distance calculator was used for both CMOS FPGA design as presented in this chapter, as well as the proposed MsFPGA design as described throughout in this dissertation. Using the two technologies, an exactly same pipeline is designed with the arithmetic blocks – subtractor, square operator, adder, comparator and multiplexers. Since CMOS is the state-of-the-art technology,

the ED pipeline was additionally designed using CMOS, so that a comparative performance analysis against the proposed memristive-CMOS hybrid design is possible [102].

This dissertation work proposes a *pipelined implementation* of the Euclidean Distance calculation using the IMPLY-memristor nanowire crossbar and, for comparison with the proposed memristive variant, the architecture is also implemented as a conventional CMOS FPGA using hardware description language VHDL [97][98]. The design was simulated for functionality evaluation and synthesized for performance measurement using Xilinx Vivado 2015.2 tool version [102].

The design contains a datapath with a controller for calculating the Euclidean distance and also calculates the overall minimum distance between all vector combinations.

A complete pipeline is implemented that has several arithmetic blocks as per the Euclidean distance formula.

$$D_{new} = SQRT[(x[0] - w[0])^2 + (x[1] - w[1])^2 + (x[2] - w[2])^2 + (x[3] - w[3])^2]. \quad (2)$$

$$D_{min} = Min\ (D_{new}, D_{min}). \quad (3)$$

The motivation of this design is a pattern recognition machine which is based on a neural network based supervised/unsupervised learning algorithm. This sub-system can be used in a design, such as, SOINN, ESOINN and GAM of Shen and Hasegawa. In the implementation discussed here, the length of each vector is kept 4-bit integer number for

simplicity, while each element of the vector is kept 8-bit wide. Each element of the vector represents a pixel of an image with gray scale value between 0-255 (i.e. 2^8). For a pattern recognition example, the length of the feature vector is calculated from the size of the image. For an image size of 28x28 pixels, the length of the vector is 784 integer numbers. For each element of this 784 is again 8-bit wide. The proposed pipeline design is based on an 8-bit width vector, however, instead of a large vector length like 784 integer numbers, the vector length was kept at 4-bit integer number only. Since, vector length does not play a significant role in this pipeline design, the length was rather kept at a variable parameter in the VHDL codes. The 8-bit vector element width dictates the size of each arithmetic block in the pipeline. Therefore, vector element width is important in this case. A longer vector length only indicates longer operation time of the pipeline to complete the ED computations between two vectors [102].

The arithmetic blocks include, subtractor, Look-up Tables (LUT), adder, comparator, SISO (shift-in shift-out) registers, accumulation register and general registers.

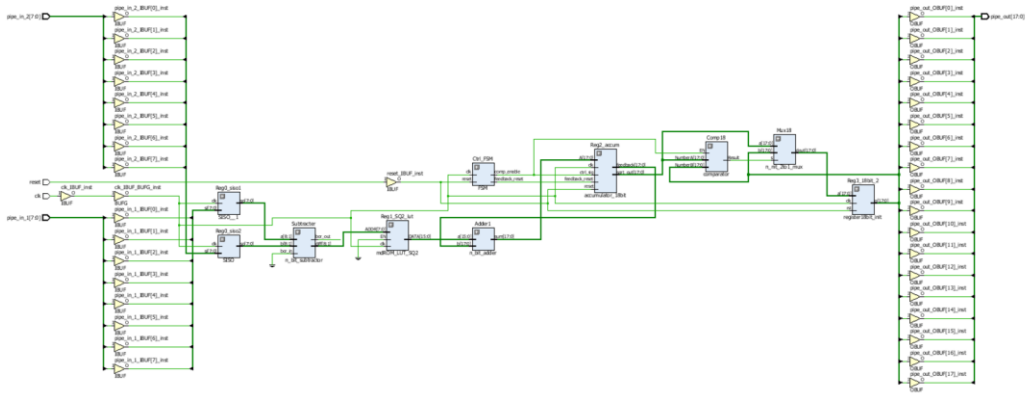


Figure 19: Complete synthesized Proposed Euclidean Distance pipeline.

The complete synthesized CMOS FPGA pipeline is shown in Figure 19. Each component of the pipeline is described separately with timing (functional) simulation and the results of the synthesis in the following section.

4.1 Detailed Implementation of Euclidean Distance Processor

Pipeline Design Blocks:

1. Serial in Serial out (SISO) registers:

The register used for this pipeline is an 8 bit SISO meaning that it shifts one element i.e. 8 bit at each clock cycle. There is an initial delay for about one clock cycle for the data to appear at the output because the SISO can take in a vector element of width eight. The vector length and width numbers are changeable as it is designed for variable "n-bit". A D-flip flop is used to build the SISO structurally, whose code is shown below along with the simulation result for sample data in Figure 20.

Code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity SISO is
generic(N : integer := 8);
   Port ( si : in  STD_LOGIC_vector(N-1 downto 0);
          clk : in  STD_LOGIC;
          so : out  STD_LOGIC_vector(N-1 downto 0));
end SISO;
```

```
architecture structural of SISO is
component dff is
generic(N : integer := 8);
   Port ( d : in  STD_LOGIC_VECTOR(N-1 downto 0);
          clk : in  STD_LOGIC;
          q : inout  STD_LOGIC_VECTOR(N-1 downto 0);
          qbar : inout  STD_LOGIC_VECTOR(N-1 downto 0));
end component;
type array_type is array (0 to N) of std_logic_vector(0 to N-1);
signal x,y : array_type;
begin
x(0)<=si;
l1: for i in 0 to N-1 generate
                d1:dff port map( d => x(i), clk => clk, q => x(i+1), qbar => y(i));
end generate;
so <= x(N);
end structural;
```

Simulation result:

The test bench will be discussed later in this chapter.

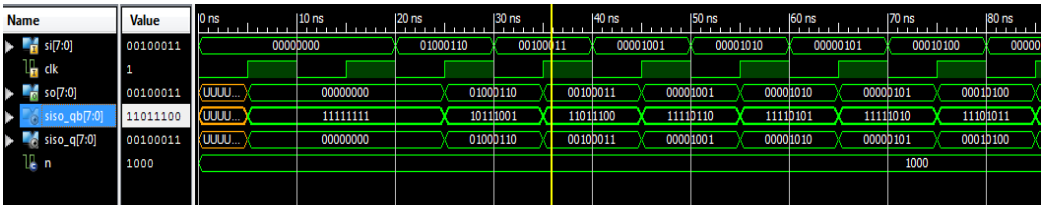


Figure 20: Simulation result SISO register.

2. Register N-bit:

This is a simple register (n-bit) as shown in Figure 21, which on getting the data at the input will give the data at the output in the next clock cycle. The register uses D-flip flops. The code is presented below and the simulation result for sample data is shown in Figure 22.

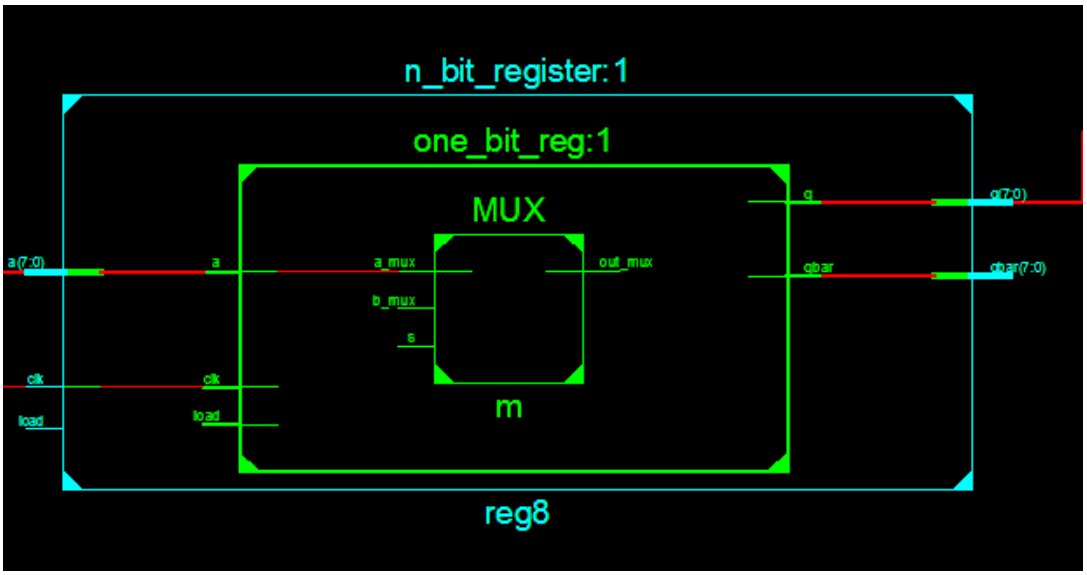


Figure 21: Structural view of register after synthesis.

Code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
entity n_bit_register is
generic(N : integer := N);
   Port ( load : in  STD_LOGIC;
          a : in  STD_LOGIC_VECTOR (N-1 downto 0);
          clk : in  STD_LOGIC;
          q : inout  STD_LOGIC_VECTOR (N-1 downto 0);
          qbar : inout  STD_LOGIC_VECTOR (N-1 downto 0));
end n_bit_register;
architecture structural of n_bit_register is
component one_bit_reg is
   Port ( load : in  STD_LOGIC;
          a : in  STD_LOGIC;
          clk : in  STD_LOGIC;
          q : inout  STD_LOGIC;
          qbar : inout  STD_LOGIC);
end component;
begin
l: for i in 0 to N-1 generate
r: one_bit_reg port map ( load => load, a => a(i), clk => clk, q => q(i), qbar => qbar(i));
end generate;
end structural;
```

Simulation result:

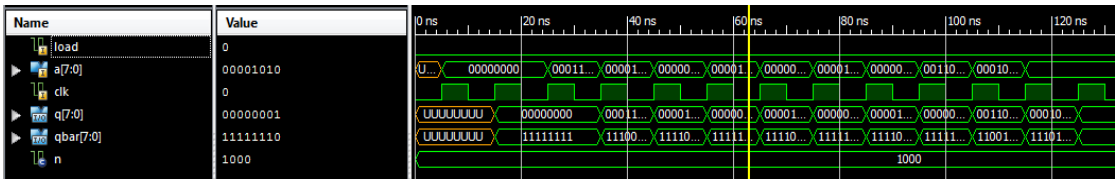


Figure 22: Simulation Result of n-bit Register.

3. Square Function:

The square LUT is a ROM that takes in an 8-bit binary number and gives a 16-bit square of the binary input as output. Xilinx Block RAMs (BRAM) were used to implement the square operation.

Originally, I planned to design the multiplier, divider, square and square-root operations using the Logarithmic Number Systems (LNS) based design [91][92]. In a Logarithmic Number System, a number x is represented as the fixed-point value $i = \log_2 x$, with a special arrangement to indicate zero x and an additional bit to show its sign. For $i = \log_2 x$ and $j = \log_2 y$ and assuming without loss of generality that, in dyadic operations, $j \leq i$, LNS arithmetic involves the following computations:

$\log_2 (x + y) = i + \log_2 (1 + 2^{j-i})$;

$\log_2 (x - y) = i - \log_2 (1 - 2^{j-i})$;

$\log_2 (x * y) = i + j$;

$\log_2 (x \div y) = i - j$;

$\log_2 (\sqrt{x}) = i \div 2$;

$\log_2 (x^2) = i * 2$.

Thus using the LNS, multiplication, division, square-root and square operations can be simplified to addition, subtraction, and shift operations.

However, later through experiments I found that the logarithmic implementation of the square operation was difficult to synthesize, and therefore, Xilinx provided block RAMs were used for the "square operator" design in this dissertation.

Since the input is 8-bit wide, the LUT in this design has 256 entries. The synthesized design and simulation waveform are presented in Figure 23 and 25 respectively. The data stored in the LUT were generated by a Python script. A small code snippet of the Square Lookup Table is shown in Figure 24.

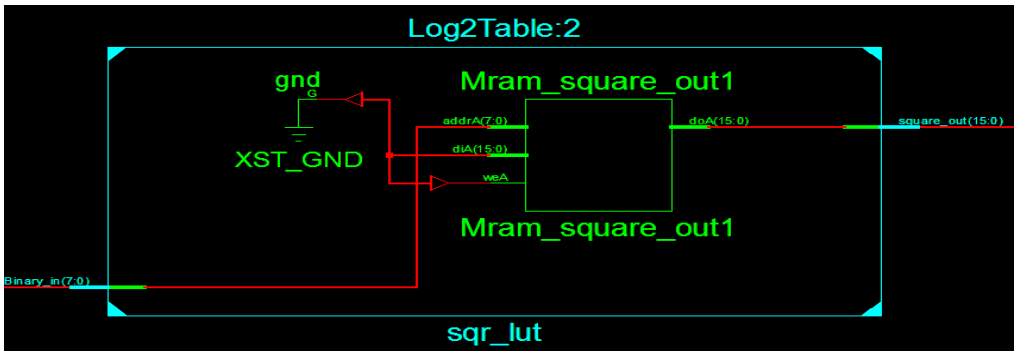


Figure 23: View of LUT after Synthesizing in Xilinx.

Code (snippet):

```
--input 8-bit binary number to output 16-bit Square number in binary
library ieee;
use ieee.std_logic_1164.all;
entity Log2Table is
  port(
```

```
        Binary_in: in std_logic_vector(7 downto 0);
        square_out : out std_logic_vector(15 downto 0));
end Log2Table;

architecture arch of Log2Table is
begin
  process(Binary_in)
  begin
    case Binary_in is
            when "00000000" => square_out<= "0000000000000000";
            when "00000001" => square_out<= "0000000000000001";
            when "00000010" => square_out<= "0000000000000100";
            when "00000011" => square_out<= "0000000000001001";
            when "00000100" => square_out<= "0000000000010000";
            when "00000101" => square_out<= "0000000000011001";
            when "00000110" => square_out<= "0000000000100100";
            when "00000111" => square_out<= "0000000000110001";
            when "00001000" => square_out<= "0000000001000000";
            when "00001001" => square_out<= "0000000001010001";
```

Sample Output from script:

```
>>>
Binary          log(Binary)             IEEE 754 format of log(binary)          Square(log(binary))     IEEE 754 format of Square(log(binary))  inv_log

001             0.0                     00000000000000000000000000000000        0.0                     00000000000000000000000000000000        000001
010             1.0                     00111111100000000000000000000000        10.0                    01000000000000000000000000000000        000100
011             1.100101011100000       00111111110010101110000000000000        11.001010111000000      01000000010010101110000000000000        001001
100             10.0                    01000000000000000000000000000000        100.0                   01000000100000000000000000000000        010000
101             10.0101001001101000     01000000000101001001101000000000        100.101001001101001     01000000100101001001101001000000        011001
110             10.100101011100000      01000000001001010111000000000000        101.001010111000000     01000000101001010111000000000000        100100
111             10.110011101010111      01000000001100111010101110000000        101.100111010101110     01000000101100111010101110000000        110001
```

Figure 24: Actual output from Python script.

Simulation result:

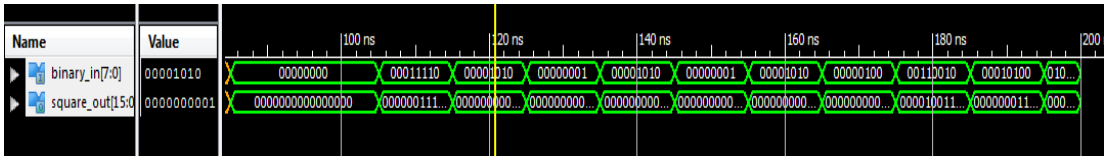


Figure 25: Simulation Result of Square Table.

4. Full adder (N-bit):

The code, synthesized circuit (Figure 26) and simulation result (Figure 27) using sample data for the N-bit full adder circuit are presented below. The adder is implemented as a ripple carry adder by full adders for N-bit parallel addition.

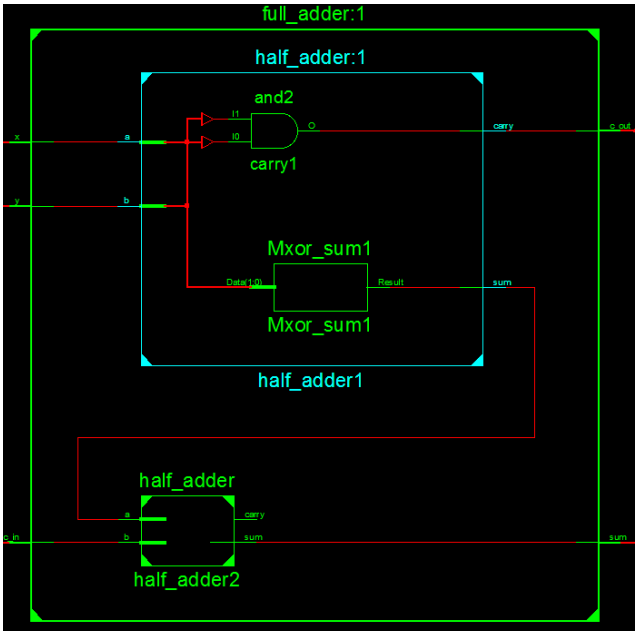


Figure 26: Structural View of adder after Synthesis.

Code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity n_bit_adder is
generic(N : integer := N);
   Port ( a : in  STD_LOGIC_VECTOR (N downto 1);
          b : in  STD_LOGIC_VECTOR (N downto 1);
          c_in : in  STD_LOGIC;
          c_out : out  STD_LOGIC;
          sum : out  STD_LOGIC_VECTOR (N downto 1));
end n_bit_adder;

architecture structural of n_bit_adder is
signal carry : std_logic_vector(0 to N);
component full_adder
       port (x    : in std_logic;
             y    : in std_logic;
             c_in  : in std_logic;
             sum  : out std_logic;
             c_out : out std_logic);
```

```
    end component;

begin
carry(0) <= c_in;
c_out <= carry(N);
gen:for I in 1 to N generate
                n_bit_adder: full_adder port map(  x => a(I),y => b(I),c_in => carry(I-
1),sum => sum(I),c_out => carry(I));
end generate;
end structural;
```

Simulation Result:

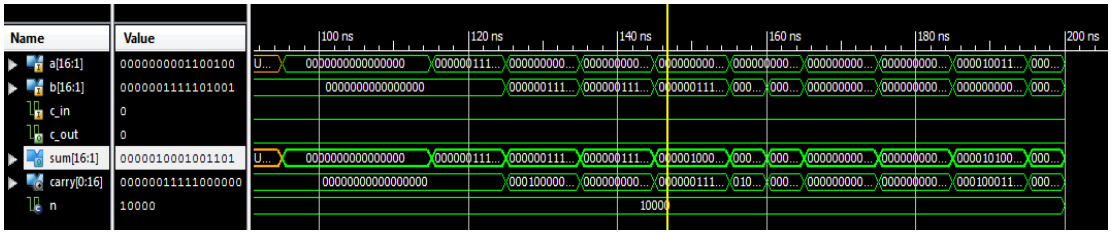


Figure 27: Simulation result of adder.

5. Accumulator (N-bit):

The N-bit accumulator uses a N-bit register to accumulate the values coming from the adder until it receives a control signal from the controller and passes it to the next block as output, meanwhile the accumulator feeds the adder. The code, synthesized circuit (Figure 28), and simulation result (Figure 29) of the accumulator are shown below. The accumulated value changes until the accumulator receives the control signal from the controller as shown in the timing simulation.

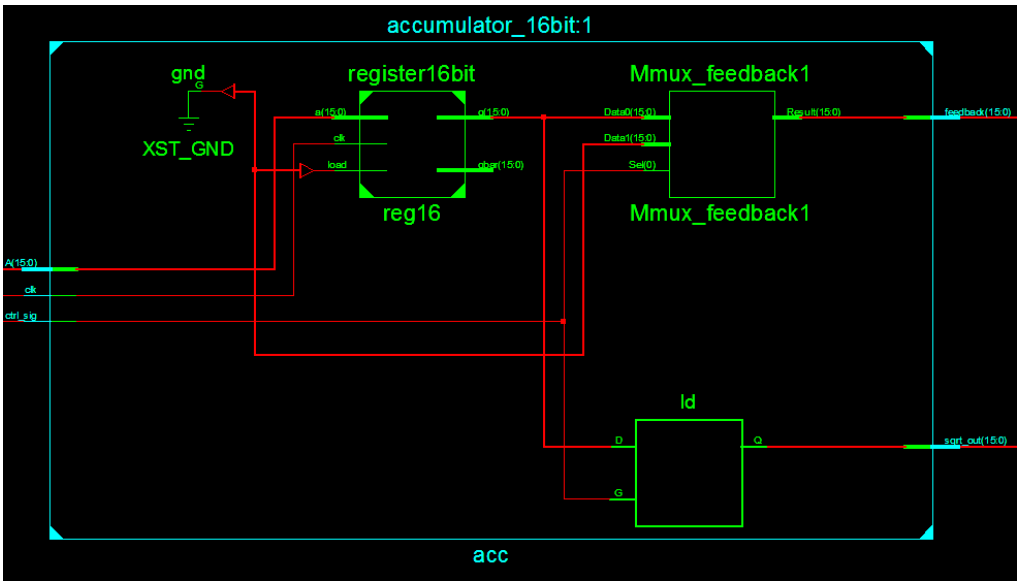


Figure 28: View of accumulator after synthesis.

Code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
entity accumulator_16bit is
generic (N: integer := 16);
    Port (A: in std_logic_vector(N-1 downto 0);
          ctrl_sig: in std_logic;
          clk: in std_logic;
          feedback: out std_logic_vector(N-1 downto 0);
                    comp_out: out std_logic_vector(N-1 downto 0));
end accumulator_16bit;
```

```
architecture Structural of accumulator_16bit is

component register16bit is
generic(N : integer := 16);
   Port ( load : in  STD_LOGIC;
          a : in  STD_LOGIC_VECTOR (N-1 downto 0);
          clk : in  STD_LOGIC;
          q : inout  STD_LOGIC_VECTOR (N-1 downto 0);
          qbar : inout  STD_LOGIC_VECTOR (N-1 downto 0));
end component;

signal reg16_out, reg16_qb: STD_LOGIC_VECTOR (N-1 downto 0);

begin
reg16: register16bit port map( load => '0', a => A, clk =>clk, q => reg16_out, qbar =>
reg16_qb);
comp_out <= reg16_out when ctrl_sig = '1';
feedback <= "0000000000000000" when (reg16_out = "UUUUUUUUUUUUUUUU" and
ctrl_sig = '0') or ctrl_sig = '1' else reg16_out;
end structural;
```

Simulation Result:

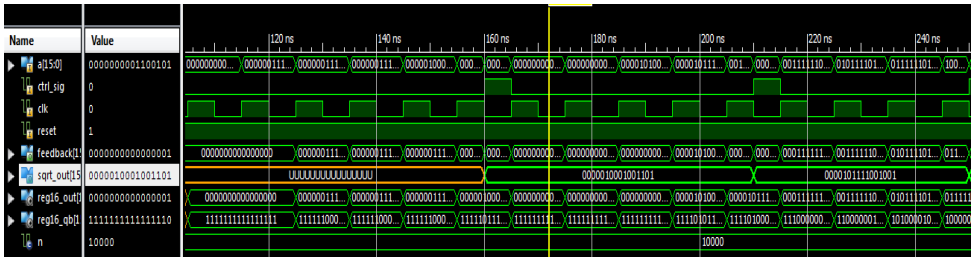


Figure 29: Accumulator simulation result.

6. Subtractor:

The subtractor used in this pipeline is different from the normal subtractor, it would always produce a positive result even if the minuend is smaller than the subtrahend, eg. 2-4 = 2. Thus the subtractor here computes the absolute difference between the two values. So, the design used in this pipeline does not output in the form of 2's complement, rather it gives the number itself, which is more suitable for the calculation purpose. The subtractor uses a comparator, and two subtractor blocks. The inputs to the unit is fed to the comparator, which compares the two values and depending on whether a is greater or not, the value of the borrow remains the same or is given a value '1'. If a is greater than b, the output of the first subtractor is obtained as output, else the output from the second subtractor is obtained as output. The top level of the subtractor is shown in Figure 30, along with the simulation results in Figure 31.

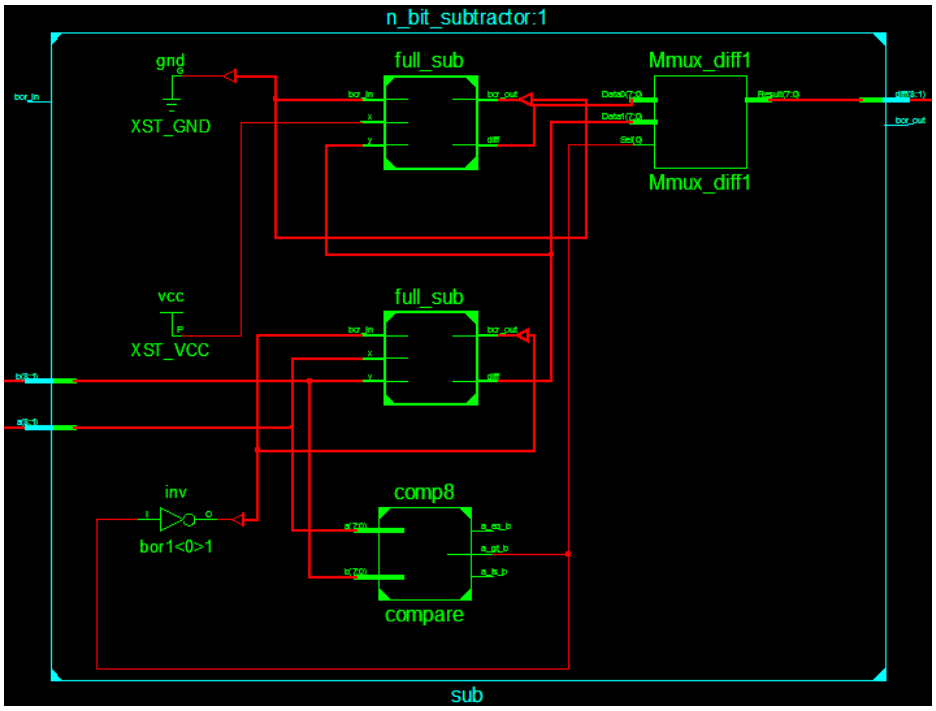


Figure 30: Subtractor design for this pipeline.

Code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity n_bit_subtractor is
generic(N : integer := 8);
    Port ( a : in  STD_LOGIC_VECTOR (N downto 1);
           b : in  STD_LOGIC_VECTOR (N downto 1);
           bor_in : in  STD_LOGIC;
           diff : out  STD_LOGIC_VECTOR (N downto 1);
           bor_out : out  STD_LOGIC);
```

```
end n_bit_subtractor;
architecture structural of n_bit_subtractor is
signal bor1,bor2 : std_logic_vector(0 to N);
signal a_eq_b,a_gt_b,a_ls_b : std_logic;
signal diff1, diff2,x_in : STD_LOGIC_VECTOR (N downto 1);
component full_sub is
   Port ( x : in  STD_LOGIC;
          y : in  STD_LOGIC;
          bor_in : in  STD_LOGIC;
          diff : out  STD_LOGIC;
          bor_out : out  STD_LOGIC);
end component;
component comp8 is
port( a,b: in STD_LOGIC_vector(7 downto 0);
                    a_gt_b: out STD_LOGIC;
                    a_eq_b: out STD_LOGIC;
                    a_ls_b: out STD_LOGIC);
end component;
begin
compare: comp8 port map( a => a, b => b, a_eq_b => a_eq_b, a_ls_b => a_ls_b, a_gt_b
=> a_gt_b);
bor1(0) <= bor_in when a_gt_b = '1' else '1';
bor2(0) <= bor_in;
```

```
x_in <= "11111111";
bor_out <= bor1(N) when a_gt_b = '1' else bor2(N);
l: for i in 1 to N generate
        n_bit_subtractor: full_sub port map(  x => a(I),y => b(I),bor_in => bor1(I-1),diff
=> diff1(I),bor_out => bor1(I));
end generate;
l1: for j in 1 to N generate
        n_bit_subtractor1: full_sub port map(  x => x_in(j) ,y => diff1(j),bor_in => bor2(j-
1),diff => diff2(j),bor_out => bor2(j));
end generate;
diff <= diff1 when a_gt_b = '1' else diff2;
end structural;
```

Simulation Result:

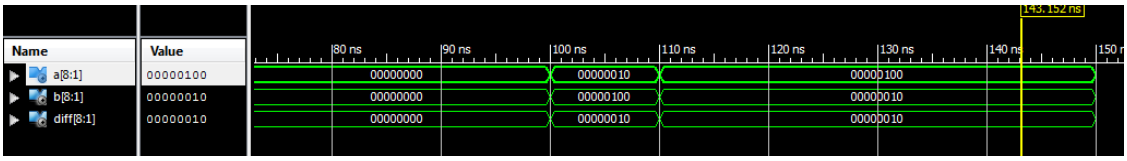


Figure 31: Simulation result of Subtractor block.

7. Controller:

The controller is an important component of the pipeline. It generates control signals for the comparator block. If comparator enable signal is high, only then the accumulator sends accumulated values to the comparator block. The block diagram from Figure 32 gives the overall view of the controller.

Block Diagram:

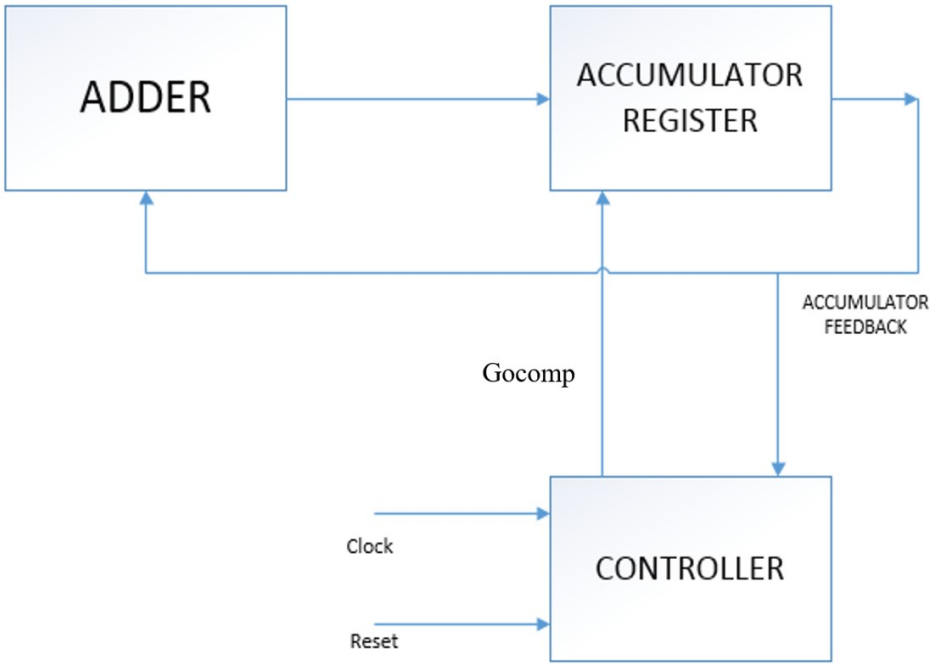


Figure 32: Controller Block Diagram.

Controller has the following inputs and outputs.

Inputs: accumulator_in (16-bit input signal from the accumulator), clock and reset

Output: comparator enable signal: comp_enable (Gocomp)

Function of the Controller:

- Controller is designed in Finite State Machine (FSM) fashion. FSM takes input from the Accumulator feedback. It has a counter inside it.
- It also has two more inputs, clock and reset.
- Based on logic inside the FSM it performs the required operations.
- After performing operations, it sends output signal comp_enable to the comparator block.

FSM:

Figure 33 shows the Finite State Machine (FSM) for the controller. This is a simple FSM block that takes input from the accumulator register and generates comparator enable signal. It has a ten-bit counter inside it. This block checks output feedback from the accumulator register and when the counter value is 784 (vector length); it generates counter enable signal to perform the comparison operation.

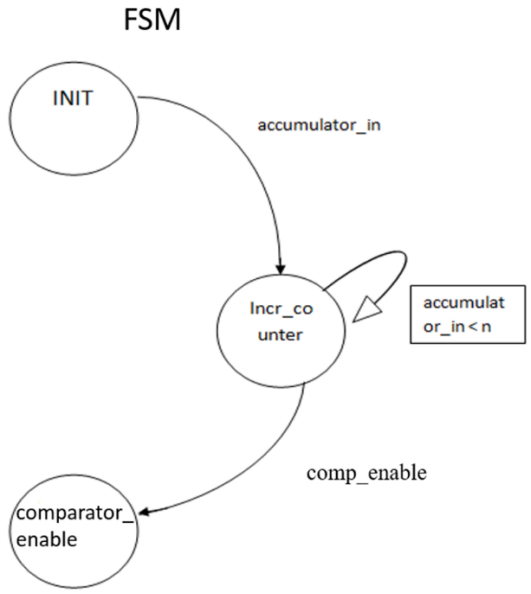


Figure 33: Finite State Machine design for the controller.

States of FSM:

INIT: This is the initialization state. It takes reset as an input. In this state comp_enable and counter are assigned to zero. When this state gets input accumulator_in, then next state will be incr_counter.

Incr_counter: This state takes input signal as accumulator_in. In this state counter keeps changing with the accumulator_in signal. When counter value is equal to 784, it points to the next step i.e the comparator_enable state.

comparator_enable: As the accumulator has gotten all adder input, this state generates comp_enable signal. It sends comp_enable signal to the comparator block to perform the comparison operation.

Code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use ieee.std_logic_unsigned.all;


entity FSM is
  Port ( reset : in STD_LOGIC;
                accumulator_in : in  STD_LOGIC_VECTOR (15 downto 0);
                clock : in  STD_LOGIC;
        comp_enable : out  STD_LOGIC
        );
end FSM;
architecture Structural of FSM is
type state_type is (Init, Incr_counter, comparator_enable);
signal counter_new: STD_LOGIC_VECTOR(9 downto 0);
signal current_s,next_s: state_type;
```

```
shared variable flag : bit;
begin
PROCESS (reset, clock)
begin
        if(reset = '0') then
                current_s <= Init;

        elsif(clock'event) then
                current_s <= next_s;
        end if;
end process ;
process (clock)
begin
        if(current_s = Init) then
                counter_new <= "0000000000";

        elsif(clock= '1' and clock'event and current_s = Incr_counter and flag ='0') then
                counter_new <= counter_new + 1;
                flag := '1';
        end if;
end process;
process (accumulator_in, current_s)
begin
```

```
case current_s is
        when Init =>
                if(accumulator_in /= "UUUUUUUUUUUUUUUU") then
                        next_s <= Incr_counter;
                else
                        next_s <= Init ;
                end if;
when Incr_counter =>
                if(counter_new = 8) then
                        next_s <= comparator_enable;
                else
                        next_s <= Incr_counter;
                        flag := '0';

                end if;
when comparator_enable =>
                next_s <= Init;
end case;
end process;

process (clock, current_s)
begin
if(clock'event) then
```

```
        case current_s is
                when Init =>
                        comp_enable <= '0';

                when Incr_counter =>
                        comp_enable <= '0';

                when comparator_enable =>
                        comp_enable <= '1';
        end case;
end if;
end process;
end Structural;
```

Simulation Result:

Simulation result for FSM is shown in Figure 34 when the counter is 2 bits.

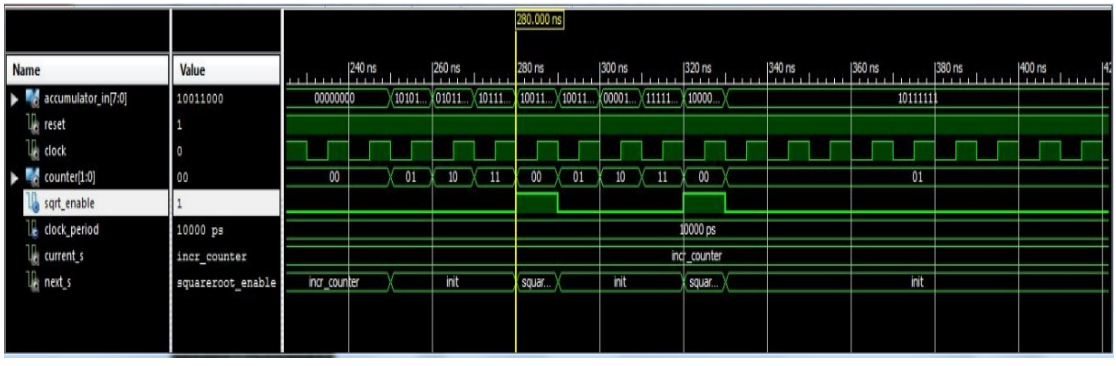


Figure 34: Simulation result of the Controller block.

Test Bench:

Testing was performed to validate the design. The test bench was generated in Xilinx Vivado. Several input values were given for the top level inputs: pipe_in_1 and pipe_in_2 and the behavior of the pipeline was tested after each clock cycle.

Simulation output after each stage is shown below:

Following inputs are given to pipeline:

```
pipe_in_1 <= "01000110";
pipe_in_2 <= "00101000";
wait for 10ns;
pipe_in_1 <= "00100011";
pipe_in_2 <= "00011001";
wait for 10ns;
pipe_in_1 <= "00001001";
pipe_in_2 <= "00001000";
wait for 10ns;
pipe_in_1 <= "00001010";
pipe_in_2 <= "00000000";
wait for 10ns;
pipe_in_1 <= "00000101";
pipe_in_2 <= "00000100";
wait for 10ns;
pipe_in_1 <= "00010100";
pipe_in_2 <= "00001010";
```

```
wait for 10ns;
pipe_in_1 <= "00000111";
pipe_in_2 <= "00000011";
wait for 10ns;
pipe_in_1 <= "01100100";
pipe_in_2 <= "00110010";
wait for 10ns;
pipe_in_1 <= "01010000";
pipe_in_2 <= "00111100";
wait for 10ns;
pipe_in_1 <= "01100011";
pipe_in_2 <= "00001001";
```

Simulation result after every clock cycle:

1. **SISO and Subtraction:**

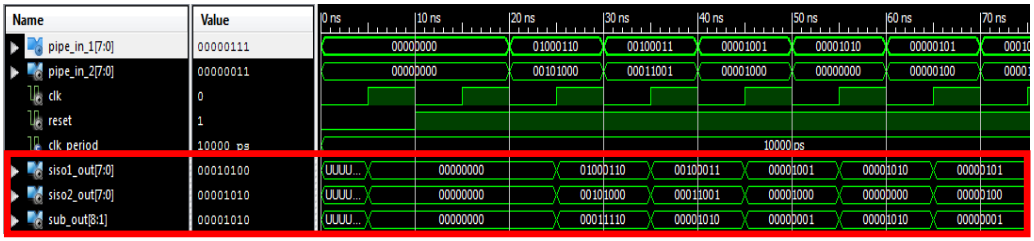


Figure 35: Testing of SISO and Subtraction unit.

As shown in Figure 35, after giving input at the first clock cycle (t=1), the output of SISO is obtained as siso1_out and siso2_out. At the same clock cycle, the subtraction operation takes place. Here we have taken pipe_in_1 and pipe_in_2 as 01000110 and

00101000 respectively. We get same values at siso1_out and siso2_out. The absolute difference of these two numbers is 00011110. Similarly, every next cycle the subtractor fetches new input values and performs shift and subtraction operation.

3. Square operation:

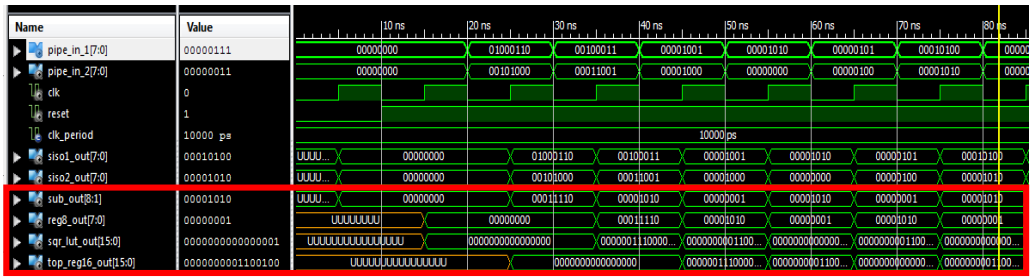


Figure 36: Testing of the Square LUT.

Figure 36 shows after performing the subtraction, at the next clock cycle t2, 8-bit register holds output value of the subtraction operation. Here value at reg8_out is 00011110. At the next clock cycle t3, square unit performs the squaring operation. Output of the sqr_lut_out is 0000001110000100. At t4, the 16-bit register, top_reg16_out gets the value of squaring unit i.e 0000001110000100.

4. Addition and accumulation:

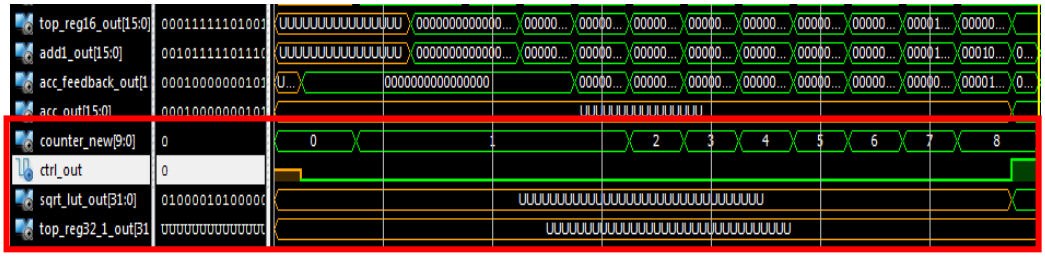


Figure 37: Testing of the addition and accumulation.

As presented in Figure 37, at t4, a 16-bit register value gets loaded into the adder. At t5, the adder adds the squared value of the next input with the older value. The adder performs addition operation untill the ctrl-out signal is low. Depending upon the counter value, controller asserts the ctrl_out signal. In this case, the counter value is 8 so it is high after getting the 8th input.

4. Comparator operation:

When the ctrl_out signal is asserted 'high', the accumulator sends the accumulated value to the comparator. In the next clock cycle, the comparator compares the value stored earlier in the 18-bit D_{min} register and the newly calculated distance value obtained from the accumulator unit. If the newly calculated value is smaller than the stored value in the 18-bit D_{min} register, D_{min} register value gets updated. Newly updated value of the 18-bit D_{min} register will be output of the pipeline (pipe_out).

Test bench code used for testing:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
 -- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY test_bench_pipeline IS
END test_bench_pipeline;
 ARCHITECTURE behavior OF test_bench_pipeline IS
```

```
 -- Component Declaration for the Unit Under Test (UUT)
   COMPONENT min_dist_pipeline
   PORT(
        pipe_in_1 : IN  std_logic_vector(7 downto 0);
        pipe_in_2 : IN  std_logic_vector(7 downto 0);
        clk : IN  std_logic;
        reset : IN  std_logic;
        pipe_out : OUT  std_logic_vector(17 downto 0)
       );
   END COMPONENT;
  --Inputs
   signal pipe_in_1 : std_logic_vector(7 downto 0) := (others => '0');
   signal pipe_in_2 : std_logic_vector(7 downto 0) := (others => '0');
   signal clk : std_logic := '0';
   signal reset : std_logic := '0';
--Outputs
   signal pipe_out : std_logic_vector(17 downto 0);
-- Clock period definitions
   constant clk_period : time := 10 ns;
BEGIN
-- Instantiate the Unit Under Test (UUT)
   uut: min_dist_pipeline PORT MAP (
         pipe_in_1 => pipe_in_1,
```

```
        pipe_in_2 => pipe_in_2,
        clk => clk,
        reset => reset,
        pipe_out => pipe_out
      );
-- Clock process definitions
  clk_process : process
  begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
  end process;
 -- Stimulus process
  stim_proc: process
  begin
    -- hold reset state for 100 ns.
-- wait for 100 ns;
        reset <= '0';
        wait for 10ns;
        reset <= '1';
        wait for 10ns;
        pipe_in_1 <= "01000110";
```

```
pipe_in_2 <= "00101000";
wait for 10ns;
pipe_in_1 <= "00100011";
pipe_in_2 <= "00011001";
wait for 10ns;
pipe_in_1 <= "00001001";
pipe_in_2 <= "00001000";
wait for 10ns;
pipe_in_1 <= "00001010";
pipe_in_2 <= "00000000";
wait for 10ns;
pipe_in_1 <= "00000101";
pipe_in_2 <= "00000100";
wait for 10ns;
pipe_in_1 <= "00010100";
pipe_in_2 <= "00001010";
wait for 10ns;
pipe_in_1 <= "00000111";
pipe_in_2 <= "00000011";
wait for 10ns;
pipe_in_1 <= "01100100";
pipe_in_2 <= "00110010";
wait for 10ns;
```

```
        pipe_in_1 <= "01010000";
        pipe_in_2 <= "00111100";
        wait for 10ns;
        pipe_in_1 <= "01100011";
        pipe_in_2 <= "00001001";
-- insert stimulus here
        wait;
  end process;
END;
```

Output of each arithmetic block of the proposed synthesized Euclidean Distance (ED) pipeline: the outputs of the subtractor, square, adder, accumulator, and comparator were tested separately. Also the complete ED pipeline was tested with different input values. The value of the 18-bit minimum distance register (D_{min} register) gets updated if and only if the newly calculated value (D_{new}) is smaller than the previously stored value. One of the original tasks was to calculate the minimum distance between a number of different vectors, which was also achieved correctly.

4.2 **Simplified Euclidean Distance (ED) Pipeline**

This dissertation proposed a hardware implementation of the ED calculation as a *pipeline design [102]*. Because, for this application, the goal was only to make comparisons between distances rather than calculating them, the implementation of the square-root function in the Euclidean Distance calculation was ignored. The pipelined datapath

consists of several arithmetic units, such as, the 8-bit SISO (shift-in shift-out) register, the 8-bit subtractor, 16-bit RAM, 18-bit adder, 18-bit accumulator register, 18-bit comparator, 18-bit multiplexer and general registers as shown in Figure 38. Xilinx provided block RAMs were used for the "square operation".

Since the element width of each vector is 8-bit, therefore, the SISO and the subtractor are both 8-bit wide. However, the square of the 8-bit subtracted number is 16-bit and thus the RAM size is 16-bit. The adder was designed as 18-bit in order to accommodate the carry bit after adding the two 16-bit numbers. Thus the following blocks in the pipeline, accumulator, comparator and multiplexer are all 18-bit wide.

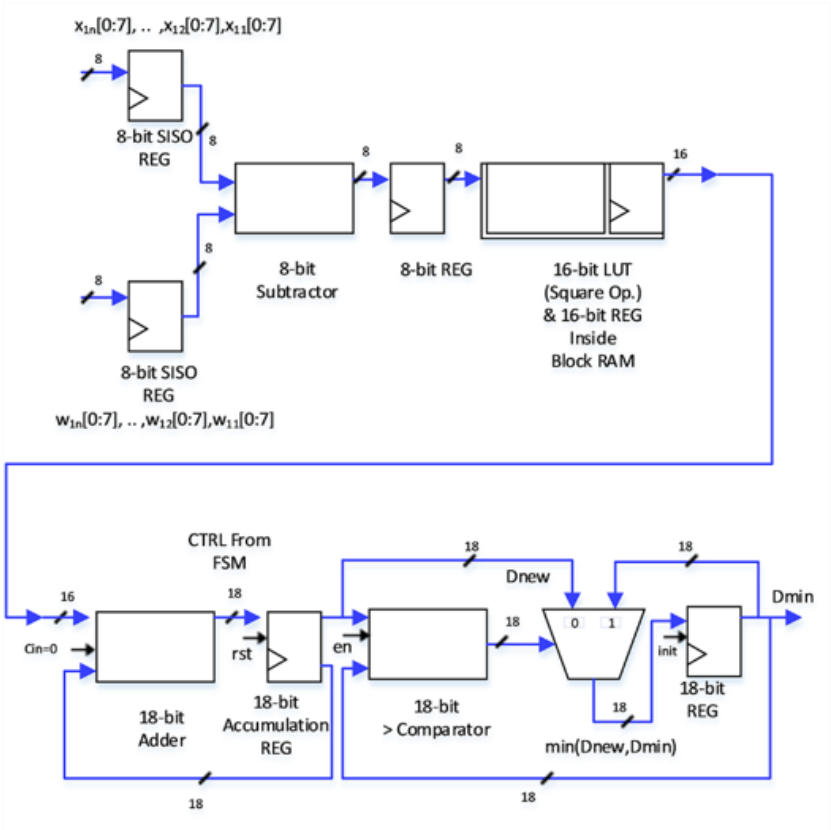


Figure 38: Pipeline Implementation of the Euclidean Distance (ED) Calculator (without square-root function) using standard CMOS FPGA [102].

The 16-bit RAM in the design is implemented as a Look-Up-Table (LUT) to function as a square operator. Two 8-bit SISO registers shift the X vector and the W vector to an 8-bit subtractor, element by element, which then sends the subtracted value of the two elements to the 16-bit RAM for performing the square operation. The squared output is fed to the adder block, and the added value is stored into the accumulator register. The accumulator register continues to accumulate the last added value and also feeds the result back to the adder until all of the vector elements have been accumulated. A finite state machine (FSM) generates the control signal to reset the accumulator register when the addition completes. The accumulated result is the newly calculated 'square of distance'. This value is compared against the previously stored square of the minimum distance and thus the new square of minimum distance is found. The comparator receives an enable signal from the FSM. This complete pipeline consists of four pipe-stages.

4.3 **Results of Xilinx Simulations and Synthesis**

The proposed design was simulated for functionality evaluation and synthesized using Xilinx® XA Vivado-2015 tool with Kintex®-7 family based smallest chip xc7k70tfbg484-3 that has a total package size of 23x23 mm^2 [102]. This FPGA class is built on a state-of-the-art high-performance/low-power (HPL) 28 nm high-k metal gate (HKMG) process technology and optimized for best price-performance with a 2X improvement compared to the previous Xilinx FPGA generations [15].

For completeness, the block by block delay numbers are presented in Table 4-1 [102]. The CMOS FPGA design was driven by a 134 MHz clock frequency, which was found to be the maximum frequency for the FPGA chip with no negative slack in the pipe-stages.

As mentioned before, in this design, the subtractor used in the pipeline is different from the normal adder-subtractor, it would always produce a positive result even if the minuend is smaller than the subtrahend, eg. 2-4 = 2. The result is always the difference of these two numbers. The design used in this pipeline does not output in the form of 2's complement, rather it gives the number itself, which is more suitable for the calculation purpose. The subtractor is built with a comparator, an 8-bit mux and two 8-bit subtractor blocks.

Table 4-1: RESULTS OF CMOS ED PIPELINE BASED ON XILINX FPGA [102].

Block	**Delay (ns)**			**Area (mm^2)**
	logic delay	**net delay**	**Total**	
8-bit subtractor	1.25	5.7	6.95	0.203
16-bit LUT Sq. RAM	3.78	0.71	4.49	0.041
18-bit adder	2.29	0.76	3.05	0.076
18-bit accumulator	2.51	0.43	2.94	0.2
18-bit comparator	3.56	1.06	4.62	0.042
18-bit mux	2.98	0.87	3.85	0.038
Pipeline Total	16.37	9.53	25.9	0.6

The estimated dynamic power for the complete design was 22.3mW and 24mW at 25% and 100% toggle rate respectively in Xilinx. The static power of the chip remains constant because all of the blocks in the FPGA are turned on regardless of their utilization.

Thus the static power of the ED pipeline design depends only on a particular FPGA type selected for comparison. Based on the total device utilization compared to the total available units in the chip, the percentage area was estimated and thus the total area was obtained; 0.904 mm^2 occupied by the ED pipeline. The details of power, delay and area estimations in Xilinx are presented in Appendix B.

5 CIRCUIT IMPLEMENTATION CHALLENGES FOR MsFPGA

5.1 About MsFPGA

This dissertation proposes a reconfigurable architecture that makes use of the stateful IMPLY logic of memristors. In this architecture, memristive crossbars operate as space-time based circuits for the datapath, and a CMOL-like datapath-memory, ***MsCMOL (Memristive stateful CMOL)*** provides reconfigurability by controlling the selection of the active nanowires in each time step. More precisely the proposed architecture has a CMOL-like datapath with memristive memories to store the pulses that reconfigure the fabric from the datapath to the logic blocks [102].

The proposed hybrid memristor-CMOS reconfigurable system level architecture, ***MsFPGA (Memristive stateful logic Field Programmable Gate Array)*** is presented in Figure 15 [102]. As mentioned earlier, the datapath, and any memory including the MsRAM of the pulse generation block of the proposed MsFPGA are designed in nanowire crossbar memristor-based technology, while the control logic- FSM controller, counter in pulse generator, multiplexers and decoder in the Merge block are designed in CMOS technology [102].

The control data in ***MsRAM*** are described with the encoding table to generate the pulses for the datapath as shown in Figure 18. This table illustrates controls for a portion of 8-bit Full-adder circuit as per Figure 9. The MsRAM controls the two select lines of a mux (control signals on the output of pulse generator). Data inputs to this mux are the voltages: for controls 00 – HiZ (High Impedance State), 10 - V_{COND}, 01 - V_{SET} and 11 -

V_{CLEAR} that are selected to control the vertical/horizontal nanowires to perform the stateful logical operations in the memristor crossbar datapath. Buffers can be inserted to prevent signal attenuation in case of long lines [102].

5.2 Comparison with Other Published Memristive FPGAs and NVM

Several concepts related to building FPGA fabric in nanotechnologies as well as some relevant components are discussed in [6][9][10][13][30]. The paper [13] by Cong et al. introduced a new idea of FPGA called MrFPGA in which all logic functions were implemented in CMOS, and only the configurations of connections were implemented by memristors playing the role of connect-disconnect switches. The authors showed advantages of this concept over previously introduced CMOS FPGAs. They concentrated on routing and compared to combinational benchmarks with standard FPGAs. In contrast, this dissertation introduces the idea of ***MsFPGA (Memristive stateful logic Field Programmable Gate Array)*** [102]. This new concept is fundamentally different from the MrFPGA architecture because memristors are used to perform all logic operations in the datapath, which leads to significant gains in area, power and delay. In addition, similar to previous work, the methodology developed here implements memories and reconfigurable connections also mostly with memristors. Proposed design is hybrid and uses some CMOS components. It is geared towards both combinational and sequential circuits, especially those with regular blocks such as iterative circuits or SIMD-like data path. But the usage of the CMOS components is insignificant compared to the previous FPGAs such as MrFPGA. To illustrate an application of proposed MsFPGA and to facilitate a comparision

with a traditional FPGA approach, an example of a pipelined implementation of a Euclidean Distance (ED) calculation is presented. The pipeline is implemented in binary logic with memristors in memristive nanowire crossbars. Memristive crossbars are perpendicular nanowires, where memristors are located at the intersections of the nanowires [8]. The goal here is to use memristors in logic design as implication gates.

MsFPGA [102] is a unique and completely different approach from published research on memristor-based FPGAs [9][10][13][30], because it allows separate programming (reconfiguration) of all- memories, logic and connections. While MrFPGA uses the general purpose combinational logic gates realized in CMOS, and therefore emphasizes connections programmability only, the proposed MsFPGA is intended for highly parallel regular architectures in which blocks are placed in abutment and horizontal connections are short. In contrast to MrFPGA, the MsFPGA methodology concentrates on logic design using stateful IMPLY gates with memristors and allows much larger logic/memory based systems with many applications. These applications include massively parallel pipelines, neural networks, SIMD-like architectures, differential equation solvers, image processors, cellular automata, and many other architectures.

Recently, a new non-volatile memory (NVM) logic architecture such as iMEMCOMP [49] was proposed by Li et al., which is different from the IMPLY gate logic. The methodology in this dissertation work is different from the iMemComp paper. The iMemComp design used Resistive Switching (RS) devices instead of memristors. The iMemComp way of realizing Boolean logic is similar to this dissertation work by adapting the idea of Stateful IMPLY logic [4] for a single bit computation, however, for multi-bit logic computation, the iMemComp method is different. The use of hybrid circuits is

different in iMemComp architecture, and the paper did not present a complete system with control, datapath and memory as was presented here. The main similarity between the iMemComp paper and this work is that both created a new type of reconfigurable logic/memory block and used the resistive way of memorizing and creating logic gates. The iMemComp paper concentrated on a new way of reconfiguring resistance values in a crossbar to realize basic logic gates, whereas, this work concentrated on system design with stateful memristors and addressed several important circuit issues. The work presented in this dissertation has several advantages over the iMemComp paper. For example, the sneak-path current is a major concern for the crossbar based design that includes, memristors, and RS cells. But the iMemComp paper did not show the complete architecture for their RS array, and therefore, it is difficult to determine if they had this problem. Sneak-path elimination is a key feature of the work presented here. In the crossbar structure, for multiple bit logic, the methodology presented here allows the movement of data from row to row, so it is possible to perform serial logic operations between rows. Whereas, the iMemComp architecture programs a look up table (LUT) for certain functions, such as a full adder and for the multi-bit design, their architecture does not support row-to-row data transfer, rather they use CMOS circuitry to transfer data, such as, carry-out. Also, the datapath in this dissertation is completely reconfigurable compared to the iMemComp LUT design. The input of proposed design in this dissertation is configured in MsRAM, while iMemComp input is the voltage signal to the decoder, which selects the row of the RS cells for the corresponding output. In general, the methodology developed here provides a wider and more flexible framework for a whole programmable system design with IMPLY-memristor based nanowire crossbars. The iMemComp focuses on the

RS switch based design for the programmable logic and does not show any of the CMOS circuitry and pulse control. Besides, it was not clear why iMemComp called logic learning rather than logic programming. However, the authors of the iMemComp paper were able to fabricate their technology. It is interesting to speculate on whether, they could fabricate a functional crossbar design.

5.3 **Proposed MsCMOL Architecture**

A memristive crossbar, which can be fabricated on top of CMOS in a back end of the line (BEOL) process, is called CMOL [6][8][14]. In "Strukov and Likharev" proposed CMOL architecture, memristors are used for storing configuration information and as such, selection of only one memristor at a time is sufficient. To address a single memory memristor in CMOL we need to individually select any of the two terminals of this memristor, one on a skewed horizontal nanowire and another one on a skewed vertical nanowire [8]. For each nanowire two CMOS decoders are necessary. These two decoders select one vertical CMOS wire and one horizontal CMOS wire. At the intersection of the vertical and the horizontal CMOS wires, the skewed nanowire is connected. This is repeated twice for vertical skewed nanowire and the horizontal skewed nanowire. Observe that in this CMOL architecture four CMOS decoders and one CMOS AND cell are used to select a single memristor, which plays the role of a single memory bit. Because, following Keukes [4], this dissertation work uses memristors in MsFPGA to execute logic operations, it is necessary to select two vertical or alternately two horizontal nanowires simultaneously for each logical operation [102]. This dissertation proposed the MsCMOL (Memristive

stateful CMOL) architecture as shown in Figure 17 [102]. Consequently, proposed approach redefines the Strukov/Likharev's CMOL architecture and its associated FPGA design methodology. In this proposed method, as in [3][4], the two vertical nanowires cross a common horizontal nanowire to execute the horizontal transfer between memristors. In addition, two horizontal nanowires can cross a vertical nanowire to execute the vertical transfer between memristors.

5.4 **Data MsRAM**

As mentioned before, the input vectors to the datapath of ED are stored in two separate Data ***MsRAMs (Memristive stateful RAM)***, which are implemented using the standard memristive nanowire crossbars or Strukov-Likharev's CMOL. Therefore, in the Data MsRAMs, only one memristor is selected at one time. The stored data is copied from these MsRAMs to the MsFPGA memristive datapath using the previously explained memristive logical transfer operation method from one memristor to another memristor located in another block [102]. The voltages used here are V_{COND}, V_{SET} and V_{CLEAR} for logical operations for the circuit designed with implication logic. The same voltages are used in MsRAMs as well. The two Data MsRAMs used in the proposed MsFPGA can also be placed within the memristive Datapath blocks for bringing the source and destination memristors closer to have better transfer. However, here it was intentionally placed outside the MsFPGA datapath as the Data MsRAMs were based on a different type of CMOL than the proposed MsCMOL [102].

5.5 **Array of 8x8 Nanowire Crossbar Blocks**

An 8×8 programmable nanowire crossbar block [102] is presented by this dissertation work. These crossbar blocks are connected horizontally as well as vertically through switches. Similar block-to-block connectivity has been discussed in previous papers [7][9], however, the approach taken here is to realize pipelined and SIMD-like datapaths as outlined in this work. Also, for the sake of comparison, this research has developed a detailed circuit that is different from previous work. An array of such small memristive 8×8 crossbar blocks can be connected through switches to form larger crossbars [7][9], and then to pipeline such crossbars, as shown in Figure 10. Since in this dissertation we assume 8-bit words in the pipeline, 8×8 crossbar blocks were assumed as cells for MsFPGA. The general methodology presented here is independent of the size of the crossbar block.

Although reference [7] and others in the past presented block to block connectivity, the method of doing the IMPLY-memristor based logic design using the space-time based symbolic notation and designing pipelined datapath circuit using the block to block connectivity concept [102] is the contribution of this dissertation. This dissertation did not invent the block-to-block connectivity for the memristor-based design, rather a method was proposed for memristor-based logic design using the space-time based concepts introduced here, where, e.g., the 8-bit iterative adder circuit uses one 8x8 nanowire crossbar implementing each row of a one-bit adder. It was believed that a square geometric shape is a better use of space than a long wire kind of shape for such cell design. The adder was designed as a single cell, but several such cells can be pipelined or executed in parallel.

Also, the proposed design performs sequential operations inside the 8x8 cell, while many such cells can operate in parallel. Therefore, using the proposed method, a large number of reconfigurable, pipelined datapaths (designed with IMPLY-memristors and controlled by pre-programmed memristive MsRAM) can operate in parallel.

5.6 **Sneak-Path Protection**

In this dissertation, sneak-path current protection was the result of the 8-bit adder design in the 8x8 crossbar. The proposed sneak-path protected design will be presented in detail in Chapter 6. Also, as shown in Figure 17, a 4-bit one-hot CMOS decoder is used to select one wire (1 row or 1 column) from a total of sixteen wires (8 rows and 8 columns) in an 8x8 nanowire crossbar block, to connect to the ground one at a time. This feature is designed to ensure that the current sinks through only one path to the Gnd and thus the sneak-path protection is enabled [102].

5.7 **Nanowire Row-to-Row Data Transfer**

A unique method was provided by this research for the row-to-row data transfer for memristors and thus switching from row-wise data transfer to column-wise data transfer [102]. As mentioned earlier, the proposed design applies V_{COND}, V_{SET} and V_{CLEAR} voltages either to a row or to a column of the 8×8 nanowire crossbar at any particular time. Therefore, in this work the row and column voltage control signals are encoded together. As presented before in Figure 17, a single bit row_sel signal is used to distinguish whether the voltages are applied to a row or to a column. When the row_sel signal is ‘1’ the voltages

are applied (through columns) to a row, and when the row_sel signal is '0' the voltages are applied (through rows) to a column. Therefore, when the row_sel signal is '1', voltages are applied to the row (through columns) from the pre-programmed MsRAM, with the column select selecting high impedance state (Hi-Z).

Proposed General Rules for the row-to-row data transfer:

For the proposed stateful IMPLY-memristor based model of combinational logic based on 8x8 nanowire crossbar blocks, the mapping of logic circuits to the detailed layout of memristors and pulses that execute operations is based on a set of rules [102]. These rules are general and apply to the squares of any size and can be also modified to blocks based on non-square rectangular arrays. We assume that in every block there are layers that are calculated sequentially one after another. For instance, the layers can correspond to cells of an iterative circuit. These layers are implemented in rows of the 8x8 nanowire block. After calculating one layer some data from it are copied to the next layer.

The following rules must be applied in order and they illustrate the space-time based transfer of values from one row to the next row.

Rule 1: In order to copy a bit from one row to another row, both bits must be located in the same column.

Rule 2: At any point of time only one wire (either a row or a column) can be discharged through ground to provide sneak-path protection. Disconnect row from Gnd and connect the column to Gnd.

Rule 3: In order to do the transfer from one row to another row, apply voltages through the rows. This is a column-wise data transfer. Stateful logic state is transferred from one row to another row.

Rule 4: In order to reset the former row memristor, V_{CLEAR} is applied in order to avoid sneak-path.

Rule 5: In order to calculate the next layer, the data needs to be transferred to a desired location in the new row. This is done by transferring bit value from one column to another column in the same row. Here, voltages are applied through columns.

Rule 6: At any point of time only one wire (either a row or a column) can be discharged through ground to provide sneak-path protection. Disconnect column from Gnd and connect the row to Gnd.

Rule 7: In order to do the transfer from one column to another column, apply voltages through the columns. This is a row-wise data transfer. Stateful logic state is transferred from one column to another column.

5.8 Proposed 8-bit Iterative Adder Design

In this dissertation, an 8-bit iterative adder design was proposed using an 8×8 memristive nanowire crossbar [102]. Each bit of the 8-bit adder is designed using one row of the 8×8 crossbar as shown in Figure 10 (one quadrant of Figure 10). All 8 rows and 8 columns are connected through a switch and a load resistor R_G to Gnd. However, the proposed design allows the connect of only one of the 16 wires (total 8 rows and 8 columns) to the ground at a time.

5.9 **Massively Parallel and Pipelined Reconfigurable Datapath**

The array of 8×8 nanowire blocks also facilitates pipelining and massive parallelism [102]. As mentioned before, larger size crossbar structure can be designed for stateful logic operations by connecting the 8×8 unit blocks through switches. An 8×8 block can be connected to another block by closing the switches, while disconnecting the switches allows various 8×8 blocks to perform stateful logical operations in parallel [7]. Also by closing the block switches a larger circuit can be created for stateful logical operations, e.g. the 16-bit RAM in our proposed ED pipeline design requires two 8×8 blocks vertically and thirty-two 8×8 blocks horizontally (or vice versa) that are connected through switches. Since by opening and closing the switches the operations of the 8×8 blocks can be controlled, a complete pipeline can be implemented row-wise as well as column-wise in this structure [102]. Many such pipelines can operate simultaneously facilitating massive single instruction, multiple data parallelism. The whole datapath of crossbar blocks introduced in this dissertation is generic. It is reconfigurable for any particular application through the pulse generation block i.e. MsRAM. Also, intrachip communication [89] can be naturally implemented using this proposed memristive crossbar datapath structure [102].

6 SNEAK-PATH CURRENT

6.1 Problems in Nanowire Crossbar Design

Sneak-path current is a critical design concern in a nanowire crossbar. Memristors are placed at each intersection of a vertical nanowire and a horizontal nanowire in a crossbar. Memristors are non-volatile memory and are able to hold a state - either logic level *low* or logic level *high* based on its resistance value at high or at low respectively.

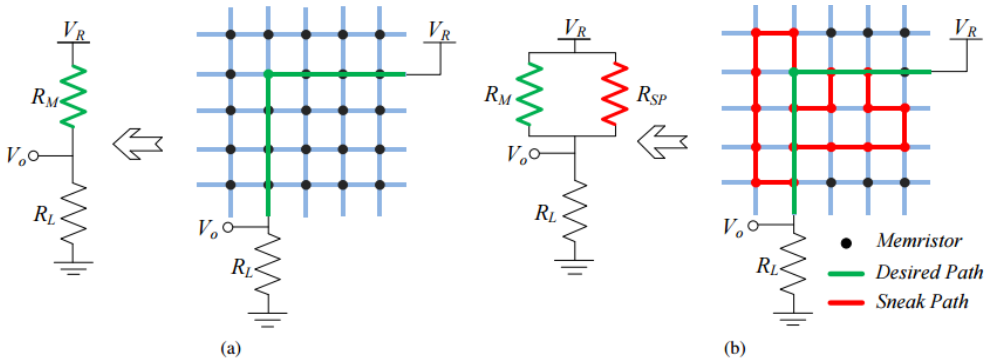


Figure 39: "The reading current path through a memristor nanowire crossbar and the equivalent circuit for (a) the ideal case where the current flows only through the target cell and (b) an example of a real case where current sneaks through different undesired path and the red ones show the effective sneak paths" [99].

Although there has been some work in creating memristors with differential forward and backward resistance, most memristors are resistors and allow current to flow either direction. As current flows through the nanowire, it can sneak through some undesired paths as shown in Figure 39 resulting in an effect which adds noise to the computation being performed by the crossbar, making it difficult to read data reliably from individual memory cells. Besides, sneak-paths also increase power consumption.

6.2 **Proposed Sneak-Path Protection**

In this chapter a sneak-path free 8-bit iterative adder design using an 8×8 memristive nanowire crossbar is presented. This innovative design methodology is a major contribution of this dissertation work.

Each bit of the 8-bit adder is designed using one row of the 8×8 crossbar. All 8 rows and 8 columns are connected through a switch and a load resistor R_G to Gnd. However, the proposed design allows the connection only one of the 16 wires (total 8 rows and 8 columns) to the ground at a time. The bit0 operation starts on row1 as shown in Figure 9 and generates carry C1 and sum S0. Upon completion of the logical operations, all of the memristors in each row are reset (cleared) using the V_{CLEAR} signals. Only the sum bits located in the eighth column of each row of the 8×8 crossbar are preserved in this method. Thus through two mechanisms: (**1**) connecting only one wire (row or column) at a time to Gnd and (**2**) resetting/clearing the memristors and forcing them to the "off" state, a complete protection from sneak-path current in the 8-bit iterative adder circuit is provided [102].

To demonstrate this approach, an 8-bit iterative adder circuit is developed for the classical design as shown in Figure 40 [35]. This circuit is redesigned with memristive nanowire crossbars using implication logic and proposed space-time based notation. The following notation is used- capital letters with respective indices are used for memristors in the datapath and the corresponding small letters are used as controls of the corresponding memristor signals from the datapath [102]. For instance, a combination of control signals a0 and a1 selects one of four possible controls used for the datapath memristor (signal) 'A'.

Similarly, control signals a00 and a01 are used for the datapath memristor (signal) A0. The carry-out signal C1 generated by the first adder bit, *bit0,* will be propagated to *bit1,* and so on. As shown in Figure 9 in Chapter 2, this 1-bit full-adder circuit is designed with three primary input nanowires for inputs A0, B0 and carry C0 and five additional working memristor nanowires.

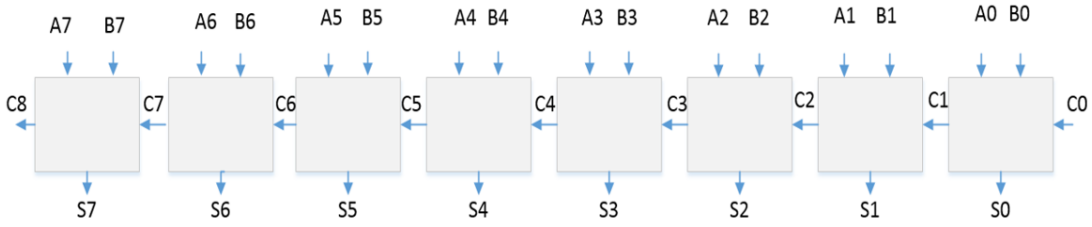


Figure 40: Classical Implementation of 8-bit Full Iterative Adder Circuit [35].

6.3 Step-by-Step Execution of Proposed 8-bit Iterative Adder

Here, the execution of the 8-bit iterative adder circuit is discussed in detail [102]. The partial encoding table of the 8-bit iterative adder design is shown in Figure 18. We start with row-wise stateful operations. The required circuit can be implemented with one 8×8 nanowire crossbar as shown in Figure 10 (in one quadrant of Figure 10). The execution is explained in four steps:

step1: The sequences of the bit0 adder operation are:

I. The primary inputs (PI) -- A0, B0, C0 are copied from storage MsRAM to the datapath row1 locations to perform bit0 operations.

Transfer the value of A0 to the memristors at the intersection of row1 and col1 in Figure 10, symbolically, (row1, col1) := A0, (row1, col2) := B0, (row1, col3) := C0.

II. Select row1, close only the row1 switch to Gnd.

III. Apply V_{COND} and V_{SET} through col1 and col2 respectively for a row-wise data transfer. A stateful logic operation will take place.

As shown in Figure 9, the carry-out bit, C1, and sum, S0, are computed after 18 micro pulses which includes all required reset operations. The carry and sum bits are saved in (row1, col7) and (row1, col8) respectively. Also, col1 through col6 are cleared to state '0' to avoid sneak-path currents.

step2: Copying of the carry bit in the current column, from the current row to the next row is presented below. Here it is demonstrated for column 7 and copying from row1 to row2.

I. Disconnect row1 from Gnd. Select col7, close only the col7 switch to Gnd.

II. Apply V_{COND} to (row1, col7) and V_{SET} to (row2, col7) for a column-wise data transfer. Stateful logic state is transferred from (row1, col7) to (row2, col7).

III. Apply V_{CLEAR} to (row1, col7).

step3: Carry bit transfer steps from one column to another in the same row. Here, voltages are applied through columns again.

I. Disconnect col7 from Gnd. Select row2, close only row2 switch to Gnd.

II. Apply V_{COND} to (row2, col7) and apply V_{SET} to (row2, col3) for a row-wise data transfer. Location of carry-out bit C1 now is at (row2, col3).

step4: This step explains the bit1 operation of the adder.

Values A1 and B1 are copied from storage MsRAM to datapath in row2 locations to perform bit1 operations. The values are transferred to the memristor locations in the row below:

(row2, col1) := A1, (row2, col2) := B1. Recall that (row2, col3) := C1.

II. Repeat the above steps from ***step1 II*** through ***step3*** for all eight rows in the 8×8 nanowire crossbar from one quadrant of Figure 10.

III. The final sum bits are located in the eighth column of each row respectively.

The total number of pulses for the 8-bit full-adder circuit operation is 165, which includes all logic, copy, and reset operations [102]. For instance, each adder requires 17 pulses to generate the sum and carry, so for 18-bit adder, 306 pulses are required for the logical operations only. However, row 3, Table 7-2 has reported 369 pulses. The source of these additional pulses were copy, reset etc. operations. Thus for the actual delay calculations in Table 7-2, all pulses required for the design were considered [102].

The proposed innovative, sneak-path free, 8-bit iterative full adder design is presented below.

t	**time step**		**Primary Input memristor**
1	**connected to GND**	**0**	**OFF memristor**
0	**Disconnected**		**V_{COND}/V_{SET} applied**

t0	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	1				0	0	0	0	0
	2	0	0	0	0	0	0	0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t1	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	1					0	0	0	0
	2	0	0	0	0	0	0	0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t2	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	1						0	0	0
	2	0	0	0	0	0	0	0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

Comment: **t0- Step 1. I. complete.**

t3	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	1						0	0	0
	2	0	0	0	0	0	0	0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t4	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	1		0				0	0	0
	2	0	0	0	0	0	0	0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t5	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	1					0	0	0	0
	2	0	0	0	0	0	0	0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t6	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	1					0	0	0	0
	2	0	0	0	0	0	0	0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t7	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	1	0					0	0	0
	2	0	0	0	0	0	0	0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t8	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	1		0				0	0	0
	2	0	0	0	0	0	0	0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t9	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	1						0	0	0
	2	0	0	0	0	0	0	0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t10	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	1						0	0	0
	2	0	0	0	0	0	0	0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t11	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	1							0	0
	2	0	0	0	0	0	0	0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t12	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	1					0		0	0
	2	0	0	0	0	0	0	0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t13	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	1		0			0		0	0
	2	0	0	0	0	0	0	0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t14	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	1		0	0		0		0	
	2	0	0	0	0	0	0	0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t15	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	1		0	0		0	0		
	2	0	0	0	0	0	0	0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t16	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	1	0	0	0		0	0		
	2	0	0	0	0	0	0	0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t17	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	1	0	0	0	0	0	0		
	2	0	0	0	0	0	0	0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

Comment: **t17- Step 1. III. complete.**

t18	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0		
	2	0	0	0	0	0	0	0		0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	1	0
	col->		1	2	3	4	5	6	7	8
t19	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0		0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	1	0
	col->		1	2	3	4	5	6	7	8
t20	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	1	0	0		0	0	0		0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

Comments: 1. t19- Step 2. III. complete 2. t20- Step 3. II. complete.

t21	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	1				0	0	0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t22	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	0	0	0	0	0	0	0	0	
	2	1					0	0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t23	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	0	0	0	0	0	0	0	0	
	2	1						0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

Comments: t21- Step 4. Bit1 operation begins.

t24	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	1						0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t25	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	0	0	0	0	0	0	0	0	
	2	1		0				0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t26	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	0	0	0	0	0	0	0	0	
	2	1					0	0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t27	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	1					0	0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t28	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	1	0					0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t29	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	1		0				0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t30	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	1						0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t31	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	1						0	0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t32	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	1							0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t33	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	1					0		0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t34	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	0	0	0	0	0	0	0	0	
	2	1		0			0		0	0
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t35	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	0	0	0	0	0	0	0	0	
	2	1		0	0		0		0	
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t36	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	1		0	0		0	0		
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t37	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	1	0	0	0		0	0		
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t38	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	1	0	0	0	0	0	0		
	3	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t39	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0		
	3	0	0	0	0	0	0	0		0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	1	0
	col->		1	2	3	4	5	6	7	8
t40	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0		0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	1	0
	col->		1	2	3	4	5	6	7	8
t41	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	1	0	0		0	0	0		0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t42	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	1				0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t43	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	1					0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t44	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	1						0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t45	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	1						0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t46	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	1		0				0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t47	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	1					0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t48	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	1					0	0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t49	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	1	0					0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t50	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	1		0				0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t51	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	1						0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t52	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	1						0	0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t53	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	1							0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t54	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	1					0		0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t55	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	1		0			0		0	0
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t56	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	1		0	0		0		0	
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t57	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	1		0	0		0	0		
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t58	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	1	0	0	0		0	0		
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t59	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	1	0	0	0	0	0	0		
	4	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t60	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0		
	4	0	0	0	0	0	0	0		0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	1	0
	col->		1	2	3	4	5	6	7	8
t61	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0		0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	1	0
	col->		1	2	3	4	5	6	7	8
t62	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	1	0	0		0	0	0		0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t63	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	1				0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t64	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	1					0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t65	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	1						0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t66	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	1						0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t67	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	1		0				0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t68	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	1					0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t69	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	1					0	0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t70	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	1	0					0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t71	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	1		0				0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t72	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	1						0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t73	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	1						0	0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t74	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	1							0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t75	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	1					0		0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t76	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	1		0			0		0	0
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t77	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	1		0	0		0		0	
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t78	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	1		0	0		0	0		
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t79	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	1	0	0	0		0	0		
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t80	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	1	0	0	0	0	0	0		
	5	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t81	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0		
	5	0	0	0	0	0	0	0		0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	1	0
	col->		1	2	3	4	5	6	7	8
t82	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0		0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	1	0
	col->		1	2	3	4	5	6	7	8
t83	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	1	0	0		0	0	0		0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t84	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	1				0	0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t85	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	1					0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t86	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	1						0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t87	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	1						0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t88	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	1		0				0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t89	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	1					0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t90	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	1					0	0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t91	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	1	0					0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t92	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	1		0				0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t93	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	1						0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t94	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	1						0	0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t95	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	1							0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t96	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	1					0		0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t97	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	1		0			0		0	0
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t98	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	1		0	0		0		0	
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t99	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	1		0	0		0	0		
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t100	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	1	0	0	0		0	0		
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t101	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	1	0	0	0	0	0	0		
	6	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t102	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0		
	6	0	0	0	0	0	0	0		0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	1	0
	col->		1	2	3	4	5	6	7	8
t103	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0		0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	1	0
	col->		1	2	3	4	5	6	7	8
t104	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	1	0	0		0	0	0		0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t105	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	1				0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t106	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	1					0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t107	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	1						0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t108	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	1						0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t109	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	1		0				0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t110	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	1					0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t111	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	1					0	0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t112	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	1	0					0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t113	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	1		0				0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t114	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	1						0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t115	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	1						0	0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t116	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	1							0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t117	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	1					0		0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t118	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	1		0			0		0	0
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t119	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	1		0	0		0		0	
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t120	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	1		0	0		0	0		
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t121	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	1	0	0	0		0	0		
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t122	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	1	0	0	0	0	0	0		
	7	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t123	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0		
	7	0	0	0	0	0	0	0		0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	1	0
	col->		1	2	3	4	5	6	7	8
t124	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0		0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	1	0
	col->		1	2	3	4	5	6	7	8
t125	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	1	0	0		0	0	0		0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t126	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	1				0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t127	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	1					0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t128	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	1						0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t129	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	1						0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t130	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	1		0				0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t131	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	1					0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t132	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	1					0	0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t133	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	1	0					0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t134	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	1		0				0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t135	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	1						0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t136	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	1						0	0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t137	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	1							0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t138	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	1					0		0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t139	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	1		0			0		0	0
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t140	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	1		0	0		0		0	
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t141	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	1		0	0		0	0		
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t142	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	1	0	0	0		0	0		
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t143	**row**	**sw**	**A**	**B**	**C**	**W1**	**W2**	**W3**	**W4**	**W5**
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	1	0	0	0	0	0	0		
	8	0	0	0	0	0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t144	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0		
	8	0	0	0	0	0	0	0		0
	sw->		0	0	0	0	0	0	1	0
	col->		1	2	3	4	5	6	7	8
t145	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0	0	
	8	0	0	0	0	0	0	0		0
	sw->		0	0	0	0	0	0	1	0
	col->		1	2	3	4	5	6	7	8
t146	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0	0	
	8	1	0	0		0	0	0		0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t147	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0	0	
	8	1				0	0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t148	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0	0	
	8	1					0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t149	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0	0	
	8	1						0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t150	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0	0	
	8	1						0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t151	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0	0	
	8	1		0				0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t152	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0	0	
	8	1					0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t153	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0	0	
	8	1					0	0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t154	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0	0	
	8	1	0					0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t155	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0	0	
	8	1		0				0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t156	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0	0	
	8	1						0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t157	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0	0	
	8	1						0	0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t158	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0	0	
	8	1							0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t159	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0	0	
	8	1					0		0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t160	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0	0	
	8	1		0			0		0	0
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t161	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0	0	
	8	1		0	0		0		0	
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

t162	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0	0	
	8	1		0	0		0	0		
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t163	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0	0	
	8	1	0	0	0		0	0		
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8
t164	row	sw	A	B	C	W1	W2	W3	W4	W5
	1	0	0	0	0	0	0	0	0	
	2	0	0	0	0	0	0	0	0	
	3	0	0	0	0	0	0	0	0	
	4	0	0	0	0	0	0	0	0	
	5	0	0	0	0	0	0	0	0	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0	0	
	8	1	0	0	0	0	0	0		
	sw->		0	0	0	0	0	0	0	0
	col->		1	2	3	4	5	6	7	8

Comment: t164- Preserved sum s0 through s8 in col8, final Cout in (row8, col7).

6.4 Benefits Brought by Proposed Sneak-path Protection Methodology

In this dissertation, a new methodology was proposed with an example of the 8-bit iterative adder design to provide protection from the sneak-path current. The proposed complete 8-bit iterative adder design was presented in section 6.3, where the step-by-step operations were shown to demonstrate the protection from sneak-path current. Using the space-time based notation, the design of IMPLY-memristor based arithmetic circuits with sneak-path current protection was presented [102]. 8x8 nanowire crossbars were used for the designs. For instance, one 8x8 nanowire crossbar was used for one 8-bit Full adder design. Each row of the 8x8 nanowire is used to implement a one-bit Full adder.

The proposed general rules for a sneak-path free design is described below:

Rule 1: In this 8x8 crossbar design, only the selected row (or column) performing logical operations is discharged through a load resistance R_G to 'gnd', while all other rows and columns in an 8x8 nanowire crossbar remain disconnected from 'gnd'.

Rule 2: After completing the logical operation, the memristor should be *reset* using the V_{CLEAR} voltage. In the proposed methodology, seven out of eight memristors are turned-off (reset) in each row in an 8x8 nanowire crossbar as shown in Figure 9.

Comment: In certain cases, this clearing operation may add insignificant delay or slight power consumption increase, yet this step is critical for providing the sneak-path protection.

Comment: The right-most column in the 8x8 crossbar preserves the "sum" bits for the 8-bit adder in an 8x8 nanowire crossbar. However, these memristors cannot contribute to the sneak-path current as there is no direct sink path available for them.

In other published papers [5][7] on memristive datapath design e.g. a full adder design using a nanowire crossbar have large number of memristors "on" at the same time in different rows and columns of the nanowire crossbar and in those designs, memristors find an alternate path to flow current to the ground causing sneak-path current. In order to provide this additional protection, a few extra timing pulses were added to the proposed design. In the space-time based notation, it is possible to have a faster design if the sneak-path current protection were not provided [102].

In the above 8-bit iterative adder design, at the beginning of the execution, A0, B0, C0 primary input data is copied to row1 locations from storage MsRAM. At t8, voltage V_{COND} and at t12 voltage V_{SET} are respectively applied to memristor C0. Therefore, memristor C0 will contribute to the static power P_{ON}. For this dissertation work, detailed power calculations were done based on the methodology presented in [6], however, it appears that this power consumption by the initial *carry bit* for proposed design is a very small number. For example, for an 8-bit iterative adder design, P_{ON} is calculated as 2.38μW with 40nm half-pitch nanowire crossbar with V_{SET} = 1.0V and 0.38μW power consumption with 8nm half-pitch nanowire crossbar with V_{SET} = 0.4V. The proposed design connects only one wire (either one row or one column) to ground at a time and therefore, there can be only one path from V_{SET} to Gnd at a time. So, for the 1-bit adder operation as shown in Figure 9, only one row is connected to the ground. Moreover, memristors are turned down

to the "reset" state through the V_{CLEAR} signal in each row after the operation is completed. Thus sneak-path protection is guaranteed in the proposed design.

This above proposed design process utilizes eight rows of the 8×8 nanowire crossbar to implement the 8-bit iterative full-adder circuit to generate all eight sum and carry signals [102]. Similarly, other iterative combinational circuits such as comparator, multiplexer, subtractor etc. have been realized for the proposed design. Algorithmic methods to realize arbitrary combinational functions with stateful IMPLY memristive logic require smart placement and partitioning of crossbar blocks. This dissertation is not related to these design automation algorithms and all designs for this research were hand-designed.

Wei Lu et al. [32] showed that memristors can be fabricated to exhibit diode characteristics. These rectifying memristors can be used to build ***converse nonimplication*** logic and may be useful to prevent sneak-path current as presented by Lehtonen [100]. However, the multi-input operation for converse nonimplication is not as useful as it is for implication logic, because, only AND-clauses result from multi-input converse nonimplication. Few important solutions proposed in the literature for the sneak paths in the memristive nanowire crossbars are discussed in [99], such as, 1T1M, 1D1M, and complimentary memristors. None of these techniques are realistic for a product design, because, 1T1M will ruin the high memristor-memory density, 1D1M will add delay and area to the design and complimentary memristors will add functional complexity to the design. Thus the sneak-path protection methodology provided for the universal implication or IMPLY logic presented in this dissertation is an important contribution.

7 PERFORMANCE STUDY OF PROPOSED MsFPGA

CMOS has become the technology of choice for its constantly shrinking chip area, faster speed and lower power consumption. However, with the scaling of device dimensions in CMOS, the voltage has stopped scaling, because, as the threshold voltage, V_T decreases, the leakage of the chip increases exponentially, increasing static power consumption. Therefore, the need for an alternate technology to continue Moore's law scaling and to meet the growing demand for lower power and faster execution motivate this work [102].

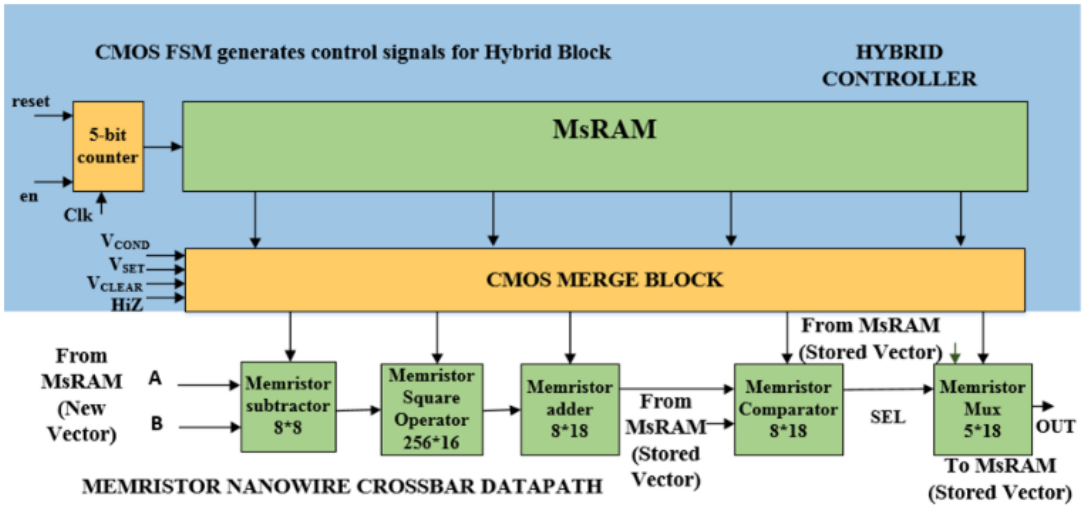


Figure 41: Pipeline Implementation of the Euclidean Distance (ED) Calculator using proposed MsFPGA, memristor-CMOS Hybrid FPGA. Color code: Green-memristor nanowire crossbar, Yellow- CMOS, Blue-hybrid circuitry [102].

In order to compare a CMOS FPGA realization of the ED calculation with the proposed MsFPGA, the pipelined version of this circuit was designed in Xilinx FPGA using standard logic synthesis. Then a pipelined ED circuit was designed with the same

functionality using the proposed MsFSMD methodology as outlined earlier. This proposed MsFPGA datapath design for functioning as the Euclidean Distance processor is presented in Figure 41.

Also, the detailed block diagram of the square operator is illustrated in Figure 42.

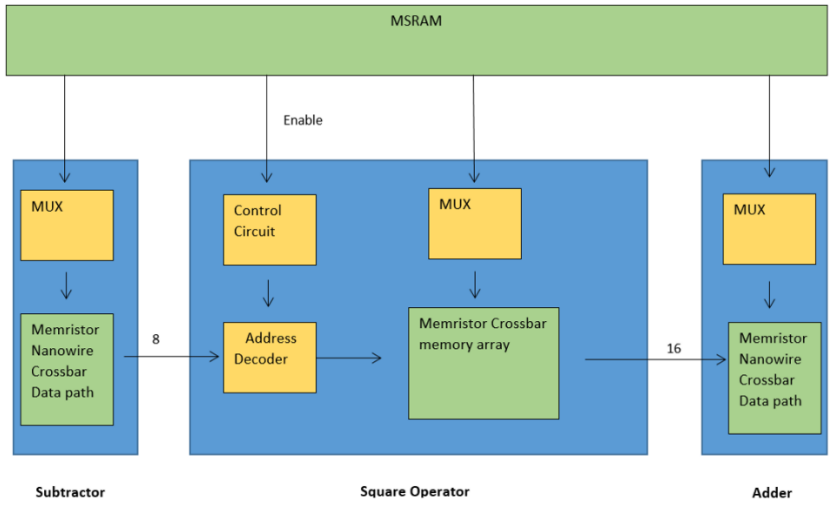


Figure 42: Block Diagram of the Square Operator. Color code: Green-memristor nanowire crossbar, Yellow- CMOS, Blue- hybrid circuitry.

The CMOS based design was synthesized and simulated using Verilog in Xilinx. Testing was performed to validate the design for correctness using test benches. Also the 8×8 nanowire crossbar in MsFPGA was simulated using PSPICE for RC delay evaluation. Besides, for verification purposes the logical behavior and the transition delay of one kind of memristive device were simulated in PSPICE.

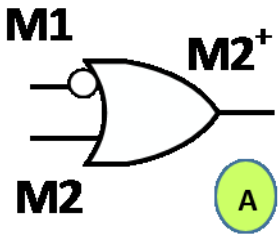
7.1 **Memristor Device and IMPLY Logic Gate**

The characteristics of memristors are unique in nature. The physical model of the memristor from [2], consists of a two-layer thin film (size D ≈ 10 nm) of TiO2, sandwiched between platinum contacts. One of the layers is doped with oxygen vacancies and thus it behaves as a semiconductor. The second, undoped region, has an insulating property.

One of the resulting properties of memristors and memristive systems is a I-V hysteresis curve on application of a sinusoidal signal [1][2][3][4][16]. In the case of linear elements, in which memristance M is a constant, it is identical to the resistance. However, if M is a function of charge, q, it yields a nonlinear circuit element [2]. For a current-controlled memristive system, the input is the current i(t), the output is the voltage v(t), and the slope of the curve represents the electrical resistance. The change in slope of the pinched hysteresis curves demonstrates switching between different resistance states which is a phenomenon central to resistive RAM (ReRAM) and all other forms of two-terminal resistance memories [1][2][40]. At high frequencies, memristive theory predicts the pinched hysteresis effect will degenerate, resulting in a straight line, representative of a linear resistor [1][2].

Two memristors can be used to perform implication with one pulse. Memristors act as a switch with two states – R_{ON} and R_{OFF}, where, R_{ON} = state 1, R_{OFF} = state 0. Voltage drop in M1 affects voltage drop in M2. M1 is input memristor and M2 is input/Output memristor as shown in Figure 43.

M1 → M2 = M1′ + M2



Case	M1	M2	$M2^+$
1	0	0	1
2	0	1	1
3	1	1	1
4	1	0	0

B

Figure 43: IMPLY Logic Gate A. Symbol B. Truth Table [3].

Different gates can be constructed using memristors by applying appropriate voltages at the memristor terminals. Implication and Inhibition gates are two fundamental gates through which we can realize other logic functions like NOT, AND, OR etc. This dissertation focuses on the functioning of a stateful implication gate. Three main voltages are required for constructing logic gates using memristors - V_{SET}, V_{COND} and V_{CLEAR} voltages. V_{SET} and V_{COND} voltages are applied to switch on the memristor by lowering the resistance. The V_{CLEAR} voltage is applied to Turn off the memristor by increasing the resistance to the maximum value.

Figure 44 shows two memristors M1 and M2 which are connected together with the load resistor R_G to form an implication gate. The value of load resistor R_G is selected in such a way such that $R_{ON} < R_G < R_{OFF}$. Inputs are given to both M1 and M2 memristors in the implication gate and the output is measured for the change in memristance/resistance of the M2 memristor.

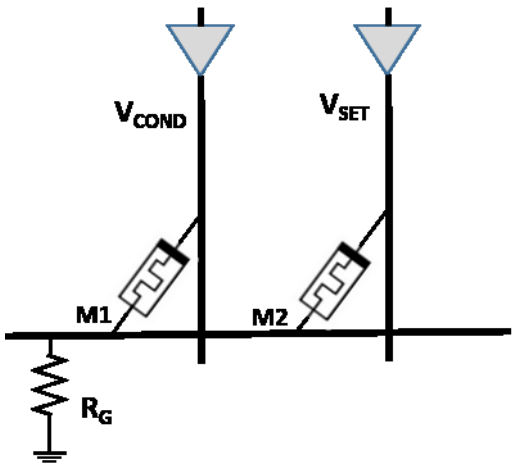


Figure 44: Implication (IMPLY) Logic: Realization with Two Memristors M1 and M2 [3].

By applying a proper voltage at the memristor terminals, we can have a memristor switch between the ON (logic '1' or close) and OFF (logic '0' or open) states. For the implication gate to work properly, it is required to apply two different voltages, V_{COND} and V_{SET} to M1 and M2 respectively. V_{SET} has a higher magnitude than V_{COND}. When M1 = 1, then voltage at R_G is approximately V_{COND}. The voltage on memristor M2 is approximately $V_{SET} - V_{COND}$. This minimum voltage is sufficient to maintain the logic state of M2. On applying V_{SET} to M2, M2′s resistance would drop close to R_{ON} and it would be set to logic 1. If both V_{SET} and V_{COND} are applied together, the current state of memristor M1 would influence the next sate of memristor M2. If M1 and M2 are both 0 in the current state, the resistance of M2 will reduce and M2 will be set to 1 in the next state. If M1 = 0 and M2 is conducting, M2 will remain high in the next state. If the resistance of M1 is low and M2′s resistance is near R_{OFF} in the present state, then the output of M2 remains low in the next state. If resistance of both M1 and M2 is high in the current state, the output of M2 remains high in the next state.

Generally used values in simulations from the literature [5] are, R_{ON} = 100Ω, R_{OFF} = 10KΩ, V_{SET} = 1.0V, V_{COND} = 0.5V, and V_{CLEAR} = -1.0V. The delay of the implication gate is measured by the time required to apply V_{SET} and V_{COND} until the logic state of M2 reaches the desired state.

7.2 Nanowire Crossbar PSPICE Simulations

For this dissertation, the memristor nanowire crossbar PSPICE [29] simulation model was created using OrCAD PSPICE software as shown in Figure 45 [102]. The simulation model represents the wire RC segments of one row and eight columns in order to simulate the 8x8 crossbar network. The fringe capacitance of the memristor device is also shown in the figure, which is placed between one column and one row (Since memristors are placed at the intersection of one row and one column).

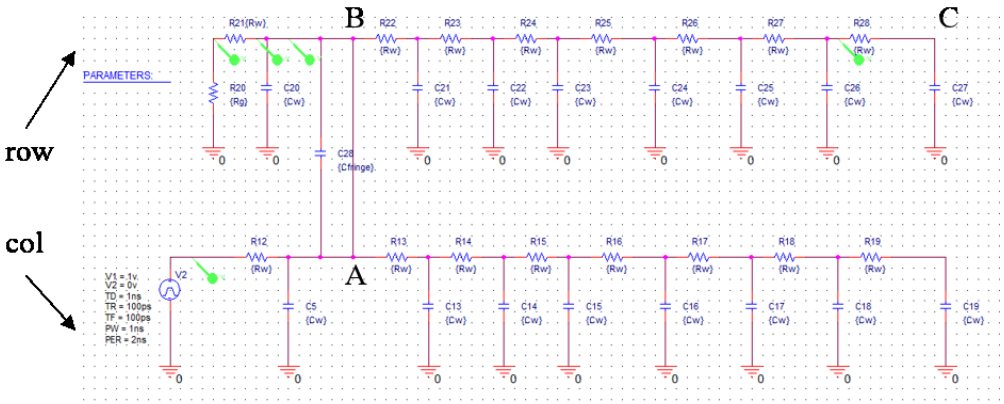


Figure 45: PSPICE Simulation Model for 8×8 nanowire crossbar [102].

Through this simulation, the nanowire *wire (RC) delays* were measured in PSPICE for V_{SET} = 1.0V, R_G = 5kΩ and nanowire half-pitch = 40nm. Nanowire 40nm half-pitch was chosen

per [20]. The load resistor R_G is connected to the row nanowire and shown at the left side of Figure 45. Also, V_{SET} is supplied through column 1 in Figure 45. The points A and B are actually the same point and represents the point of intersection of column 1 and row. The current flows from column 1 through R_G load resistor to Gnd in Figure 45. This is the nearest current path to sink through the load resistor and therefore, represents the shortest wire delay. A connection between point A and point C would thus represent the longest wire delay. Simulations were performed for each column current source to sink path and wire delays were measured. In each case the wire delays were less than 2fs, which is negligible compared to the memristor device delays reported by various memristor models [17][5][18]. Figure 46 shows RC delay measurement of the crossbars.

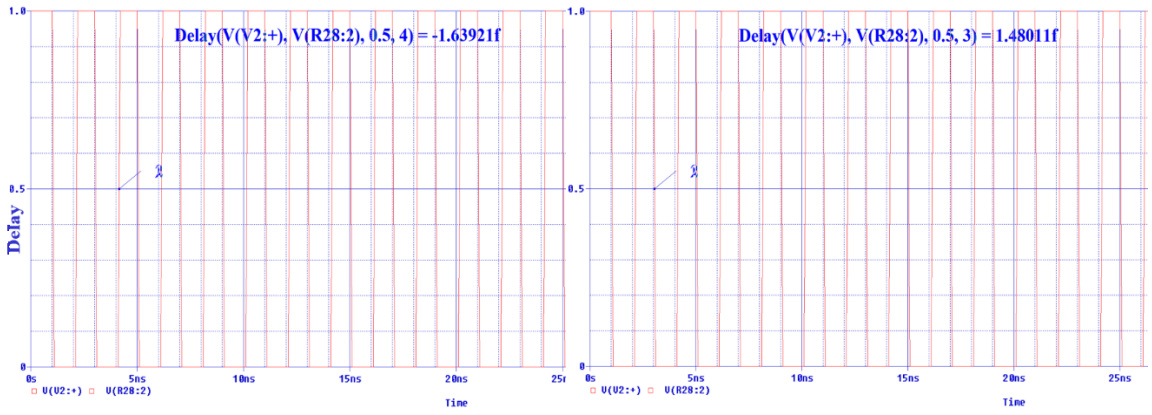


Figure 46: PSPICE Simulation Results for 8×8 nanowire crossbar; RC Delay measurement in PSPICE for VSET= 1.0V, Nanowire half-pitch=40nm. Results from Two Separate Runs are shown side-by-side [102].

7.3 **Performance Study**

7.3.1 Memristor Device Delay

Behavioral models of memristive devices can provide an overview of the expected characteristics. However, to determine the actual circuit performance we need a model that would contain various process-dependent parameters, through which the devices can be tweaked to optimize the design for performance improvement, such as delay reduction [17]. Several papers have reported the delays of memristors for logical transfer operations using behavioral models. Kvatinsky et al. [5] reported the delay of the memristive implication gate to be 397.1ns using the ThrEshold Adaptive Memristor (TEAM) model with a TiO_2 based memristor. Torrezan *et al.* [18] showed that the set and reset operations were successfully performed in the TaO_X (Tantalum Oxide) memristor using pulses with durations of 105 and 120ps, respectively. Mazady et al. [17] recently reported a promising work based on ZrO_2 memristor. They claimed their memristor model to be the only one so far not based on a behavioral memristor model, but rather based on the underlying physics of the device, which allows the optimization of circuit performance. They estimated the delay of the ZrO_2 memristor to be only 6.8ps, which is due to a very high mobility of 370 cm2/V-s of ZrO_2 with a resistivity of 1.33×1013 Ω-cm for the insulating material.

Since the research goal of this dissertation is methodology development for circuit and system design and not device modeling, logic transition delay numbers from published research on various memristor models [17][5][18] were used for performance evaluations of the proposed memristor-CMOS hybrid ED pipeline design.

7.3.2 Memristor Nanowire Crossbar Delay Evaluation

For this work, a simulation model was built to evaluate the RC delay of the 8×8 nanowire crossbar [102]. The wire resistance and wire capacitance values were calculated [6] to use in the simulation model. Also a fringing capacitance was added for the device. Simulations were performed using 1.0V V_{SET} voltage for the 40nm half-pitch nanowire crossbars [20] as shown in Figure 45. Simulation results showed that even for the worst case, which is the farthest segment from the load resistor R_G, the RC delay was only ~2fs, and thus it is negligible when added to each transition delay in Table 7-2 [102]. The nanowire model mentioned above can be tweaked further for more accuracy through adjusting the wire resistances and capacitances, however, the results will not be significantly different in order to make any change to the overall pipeline delay.

7.3.3 Power Estimation of Memristor-Nanowire Design

The three possible sources of power consumption for the memristive nanowire crossbar design are listed below:

1. Static power P_{ON} due to current I_{ON}
2. Static power P_{LEAK} due to leakage current through nanodevices in their OFF state. Sneak-path current is considered a leakage type of power consumption [101].
3. Dynamic power P_{DYN} due to the recharging of nanowire capacitances.

The power calculations were performed based on [6]. For convenience of the readers the calculation process with equations [6] are presented below.

Static Power, P_{ON} is expressed as,

$$P_{ON} = \frac{V_{DD}^2}{2R_{ser}}$$

$$R_{ser} = \frac{R_{ON}}{D} + 2R_{wire} + R_G$$

Therefore,

$$P_{ON} = \frac{V_{DD}^2}{2(\frac{R_{ON}}{D} + 2R_{wire} + R_G)}$$

Static Power, P_{LEAK} is expressed as,

$$P_{leak} = \frac{MV_{DD}^2}{2R_{OFF}/D}$$

Dynamic Power, P_{DYN} is expressed as,

$$P_{dyn} = \frac{C_{wire}V_{DD}^2}{4\tau}$$

The value of the parameters in the above equations are furnished below:

D (parallel connection of memristive devices to latching switch) = 8

M (closed switches in parallel) = 2

$V_{DD} = V_{SET}$

$R_{ON} = 100\Omega$

$R_{OFF} = 10k\Omega$

τ $(total\ circuit\ delay)$,

The total circuit delay was calculated using the proposed space-time based notation for the Euclidean Distance Pipeline based on memristor device models from three published research [17][5][18].

$$C_{wire} = C_{wire}/L * L_{wire} = (C_{wire}/L) * 7 * Distance_{nano}$$

$$R_{wire} = \rho \frac{L}{A} = \rho_0 * \left(1 + \frac{l}{F_{nano}}\right)\frac{L}{A}$$

$$= 20\Omega - nm * (1 + 10nm/F_{nano}) \frac{Distance_{nano}}{F_{nano}^{\ 2}}$$

l = electron mean-free path = 10 nm, which is typical for good metals at room temperature

$\rho_0 = 20\ \Omega - nm$

variables:

$$\tau: total\ circuit\ dealy$$

$$Distance_{nano}: distance\ between\ nano\ wire$$

$$F_{nano}: half\ pitch\ of\ namano\ wires$$

$$C\_wire/L: obtained\ from\ below\ graph\ based\ on\ various\ nanolayer\ separation$$

The wire capacitance values were generated using the well-known FASTCAP code as mentioned in [6], for the crossbar structure in which both width and thickness of the nanowire, as well as the horizontal distance between the wires, were assumed to be all equal to *F*nano, while the vertical distance between two layers was varied between 2 to 4 nm. These wire capacitance values were used in the PSPICE simulations and for power estimation for this research.

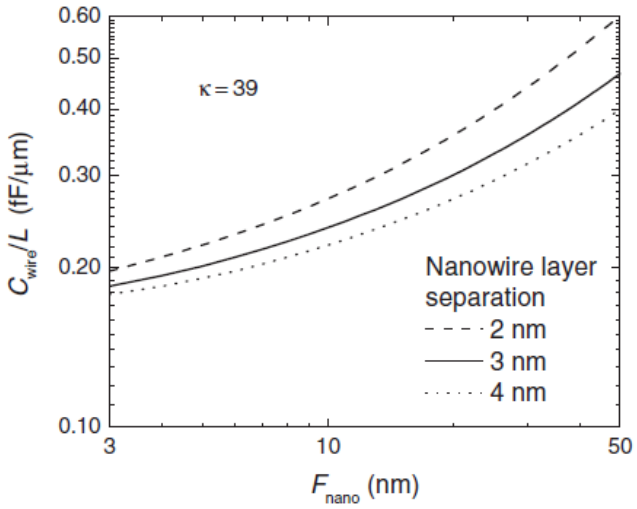


*Figure 47: **C_{wire}/L** Calculation [6].*

P_{DYN} Calculation:

Detailed power calculations were performed based on both 40nm half-pitch [20] and 8nm half-pitch [6] ***nanowire (nw)*** crossbars at two different V_{SET} voltages. Therefore, the below four combinations [102] were used for the P_{DYN} calculation. Also, C_{wire} values were calculated based on 2, 3, and 4nm separations as shown in Figure 47.

1. 40nm nw width, spacing; V_{SET} = 1 V: calculate P_{DYN} using delay, τ from [17][5][18].
2. 8nm nw width, spacing; V_{SET} = 1 V: calculate P_{DYN} using delay, τ from [17][5][18].
3. 40nm nw width, spacing; V_{SET} = 0.4 V: calculate P_{DYN} using delay, τ from [17][5][18].
4. 8nm nw width, spacing; V_{SET} = 0.4 V: calculate P_{DYN} using delay, τ from [17][5][18].

As presented in Figure 48, detailed dynamic power calculations [6] were performed based on both 40nm half-pitch [20] and 8nm half-pitch [6] nanowire crossbars. Figure 48 also shows the total dynamic power P_{DYN} consumed by the complete memristor-based pipeline for the three types of memristor devices that were used for the device delay

calculations [Table 7-2]. The worst case P_{DYN} is 9.63nW running at a V_{SET} voltage of 1V consumed by the pipeline for the 40nm half-pitch nanowires.

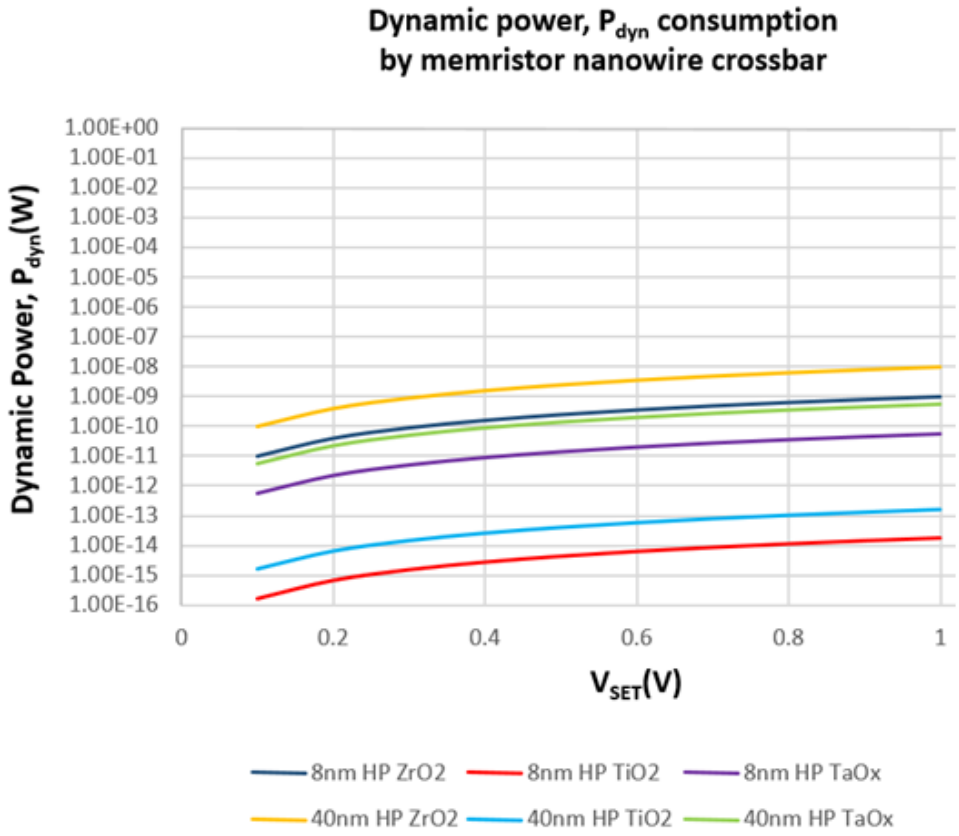


Figure 48: Total estimated dynamic power, Pdyn of the proposed memristor based complete pipelined datapath. Results show Pdyn consumption by various memristor device based designs [from Table 7-2] with both 8-nm and 40-nm half-pitch nanowires [102].

P_{LEAK} and Sneak-path Current:

Since memristors are resistors that conduct current constantly, sneak-path current is a concern for any memristor based design [102]. In the previous chapters, with the example of an 8-bit iterative adder circuit (Figure 9, Figure 10, and Figure 18) a methodology was proposed to provide the sneak-path current protection to the IMPLY-

memristor based design. The proposed 8-bit adder eliminates all logic sneak-path currents and minimizes power related sneak-path current [102]. In this proposed design only one row or column of the 8×8 crossbar at a time can discharge through Gnd. Moreover, memristors go through reset with the V_{CLEAR} voltage after completing the operations in each row. In the proposed 8-bit adder design only the sum bits are preserved in the eighth column of the 8×8 nanowire crossbar, while all other memristors are cleared. However, the memristors that are holding the sum bits cannot easily discharge as no direct path to Gnd is available for them. Thus this dissertation assumes that the leakage power P_{LEAK} is negligible in the proposed design. However, without any sneak-path protection, the worst case estimated leakage power would be 0.8mW [6] per pipestage. This means that the hybrid system consumes 11% less power due to the sneak-path protection provided by this methodology.

P_{ON} Calculation:

Finally, the P_{ON} power was calculated. As shown in Figure 18, in proposed methodology the primary input data is copied over from storage MsRAM to row1 of the 8×8 nanowire crossbar [102]. Therefore, as shown in Figure 9, the initial carry bit holds the memristance value for several pulses/cycles. This may cause some static power loss. However, since other protections are provided to the design, this loss is also negligible. The calculated P_{ON} power loss is 2.38µW per pipestage. Therefore, the power loss due to the initial carry bit is only (8/165) * 2.38 = 0.115µW per 8-bit adder.

Here, the initial carry bit does not participate in any transfer during the first 8 pulses, while a total of 165 pulses are required for the 8-bit adder implementation.

CMOS **P_{DYN} Calculation:**

The estimated dynamic power consumption by CMOS circuitry in the hybrid chip, per 8×8 memristive nanowire crossbar (per pipestage) is 2.25mW at 25% toggle rate and therefore, for four pipestages of the complete ED pipeline, it is approximately 9mW [102].

Thus in the proposed methodology, the total power consumption of the memristive-nanowire crossbar is negligible and the CMOS dynamic power dominates the overall power consumption of the hybrid design. The proposed hybrid MsFPGA operating at the supply voltage of 1.0V consumes ~9mW total dynamic power.

7.3.4 Memristor-Nanowire Crossbar Area Estimation

Memristors are physically located at the intersections of each horizontal nanowire and vertical nanowire in a two-layer crossbar network. Therefore, the total nanowire crossbar area (x and y dimension) required for the ED pipeline design was calculated [102]. Based on several publications on the fabrication of nanowire crossbars [20][30][31][32], the required area for the ED pipeline was estimated. For example, Borghetti et al. [20] fabricated 40 nm half-pitch memristor crossbars using nanoimprint lithography on the same silicon substrate with CMOS, for fully integrated hybrid circuits. Half-pitch is defined as half the distance between two nanowires from center to center. Therefore, the width of nanowire=40 nm, spacing of nanowire =40 nm and the center to center distance is 80nm. Also, since memristors in a crossbar are located at the intersection of each horizontal nanowire and each vertical nanowire, the ***memristor cross-sectional area*** for a 40-nm half-pitch is 40 nm x 40 nm.

Before we can estimate the design area and delay calculations, every memristors-based block in the MsFPGA system is designed with IMPLY-memristors using the space-time notation. Thus we know how many memristors or 8x8 crossbar block(s) and also how many pulses are required for designing each component or block. The number of memristors is used for the layout area estimation and the number of pulses is used for the delay calculation.

Figure 49 and Figure 50 elaborate the area calculation method for an 8x8 nanowire crossbar for the 40 nm half-pitch memristor crossbars.

X-direction distance = (Full-pitch between nanowires * number of nanowires in the middle) + (Half of two side nanowires on both ends).

Half of each nanowire = 20nm;

X-direction distance = (80 * 7) + (20) = 600 nm;

Similarly, Y-direction distance = 600 nm;

Thus the total area of 8x8 crossbar = 0.6 * 0.6 μm^2 = 0.36 μm^2

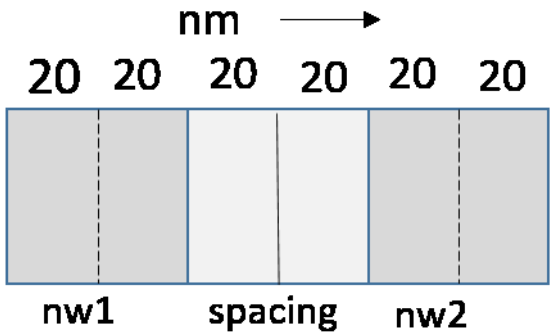


Figure 49: 40-nm Half-pitch Distance Between Two Nanowires.

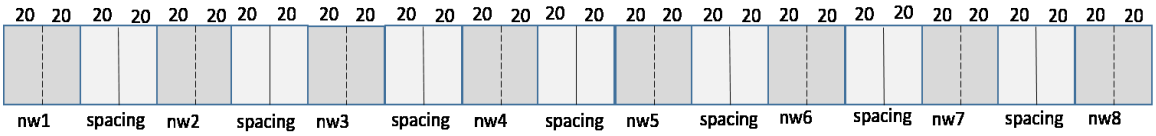


Figure 50: X-Distance Measurement for eight vertical nanowires. Total x-Distance is 0.6µm. Similarly, total y-Distance for eight horizontal nanowires is 0.6µm. Therefore, the area of an 8x8 nanowire crossbar is 0.36µm^2.

Table 7-1 shows the total calculated area breakdown for various components in the ED pipeline datapath [102]. The MsRAM areas for the subtractor, adder, comparator and multiplexer were calculated. Since the total number of micro-pulses for the complete pipeline is 1027 and the total number of control bits is 21, therefore the total MsRAM area was calculated as shown below:

MsRAM area = ((Total number of micro-pulses * Total number of control bits) * Total area for 8x8 crossbar)/Total number of memristors in an 8x8 crossbar.

Thus, the total MsRAM area for the ED pipeline is calculated as 121.31µm^2.

Table 7-1: CALCULATED AREA OF COMPONENTS OF ED PIPELINE DATAPATH [102].

Component	Memristors	Area(μm^2)
8-bit sub	8x8	0.36
LUT	256x16	23.04
18-bit adder	8x18	0.81
18-bit comp	8x18	0.81
18-bit mux	5x18	0.51
Total		25.53

Based on ref. [20], the estimated area [presented in Table 7-2] of the proposed ED pipeline datapath is 25.5μm^2, with corresponding MsRAM area in Pulse Generator is 121μm^2. Therefore, the total area consumed by the complete ED pipeline is 146μm^2. However, with the use of 8nm half-pitch nanowires [6], the total area requirement of the same above mentioned memristor-based ED pipeline is only 5.9μm^2. Besides, the CMOS circuitry in the hybrid MsFPGA consumes 0.32mm^2 area as estimated. Therefore, the area of the hybrid MsFPGA design is dominated by the CMOS components [102]. As the memristor crossbar technology matures, more components can be converted from CMOS to memristors and thus these components can be moved to the memristor layer from the CMOS layer [102].

Table 7-2: CALCULATED DELAY AND AREA FOR ED PIPELINE USING IMPLY-MEMRISTIVE NANOWIRE BASED MsFPGA DESIGN [102].

Block	**Micro pulse required**	**Delay Based on ZrO_2 memristor for each transfer = 6.8ps [17].**	**Delay Based on TiO_2 memristor TEAM Model for each transfer = 397.1ns [5].**	**Delay Based on TaO_x memristor for each transfer = 120ps [18].**	**Area (μm^2) Based on Ref. [20].**
		Realistic Process Model	**Behavioral Model**	**Behavioral Model**	
8-bit Subtractor	224	1.52ns	88.95µs	26.88ns	0.36
16-bit LUT RAM	35	0.24ns	13.9µs	4.2ns	23.04
18-bit Full Adder	369	2.51ns	146.53µs	44.28ns	0.81
18-bit Comparator	290	1.97ns	115.16µs	34.8ns	0.81
18-bit Multiplexer	109	0.74ns	43.28µs	13.08ns	0.51
Pipeline Total	1027	6.98ns	407.82µs	123.24ns	25.53
PG MsRAM For 5 Blocks in Pipeline	-	-	-	-	121.31

7.4 **Memristor-based Pipeline Design**

As discussed in Chapter 4, a CMOS pipelined circuit has a series of combinational-sequential alternating blocks. Memristors are non-volatile memory and act like a Finite

State Machine (FSM). Therefore, for the memristor-implication based pipeline design, memristors function as sequential elements. So, at the output of the memristor-based combinatorial block, memristors act like an asynchronous delay circuit elements and hold the data. Thus the standard registers (such as those with D Flip Flops) and their standard clock are not required for the proposed methodology as illustrated in the memristor-based pipeline design. This useful advantage makes the memristor-based design more efficient compared to the CMOS-based pipelined circuits and thus saves area, and power, and reduces the design complexity significantly. The example illustrates that the proposed design style is the best for massively parallel multiple pipeline designs which are typical for image processing, digital signal processing, control, pattern recognition, neural network emulation, data mining and similar applications, driving forces for the development of new hardware technologies and their associated design methodologies.

The ED pipeline was designed using the space-time notation for implication gates realized with memristors as presented in Figure 53 [102]. The calculated delay and area numbers for this design are presented in Table 7-2 [102]. Two behavioral models [5][18] and one process-based model [17] from the literature were used for memristor device delays and the nanowire crossbar was simulated in PSPICE for RC delay for the proposed IMPLY-memristor based ED pipeline design. The performances of the CMOS components in the hybrid design were obtained through HDL (Hardware Description Language) Verilog simulations and synthesis results using Xilinx tools [15].

8 RESULTS

Comparative Performance Analysis of MsFPGA

As presented in the previous chapters, the design and methodology of MsFPGA, which is a stateful IMPLY-memristor-CMOS hybrid FPGA was proposed in this dissertation. This dissertation work has also proposed the pipelined implementation of the Euclidean Distance (ED) Processor. The example of Euclidean Distance calculator was used for both CMOS FPGA design as well as MsFPGA design. Using the two technologies, an exactly same pipeline is designed with the arithmetic blocks − subtractor, square operator, adder, comparator and multiplexers. Since CMOS is the state-of-the-art technology, the ED pipeline was additionally designed using CMOS, so that a comparative performance analysis against the proposed memristive-CMOS hybrid design is possible.

For the convenience of the readers, Figure 51, Figure 52, and Figure 53 are presented again in this chapter. Logic components of Figure 51, CMOS FPGA ED pipeline and "virtual" registers are all included in the red polygon of Figure 52, MsFPGA showing one pipeline. Proposed MsFPGA architecture also contains memories and a pulse-generator which are problem-specific programmable blocks and are shown in Figure 52. Also, the components in Figure 52 red polygon are marked/mapped in Figure 53, memristor-CMOS Hybrid ED pipeline. The red polygon in Figure 52 is nothing but a fabric, the actual logic is configured in the MsRAM located in the Pulse Generator.

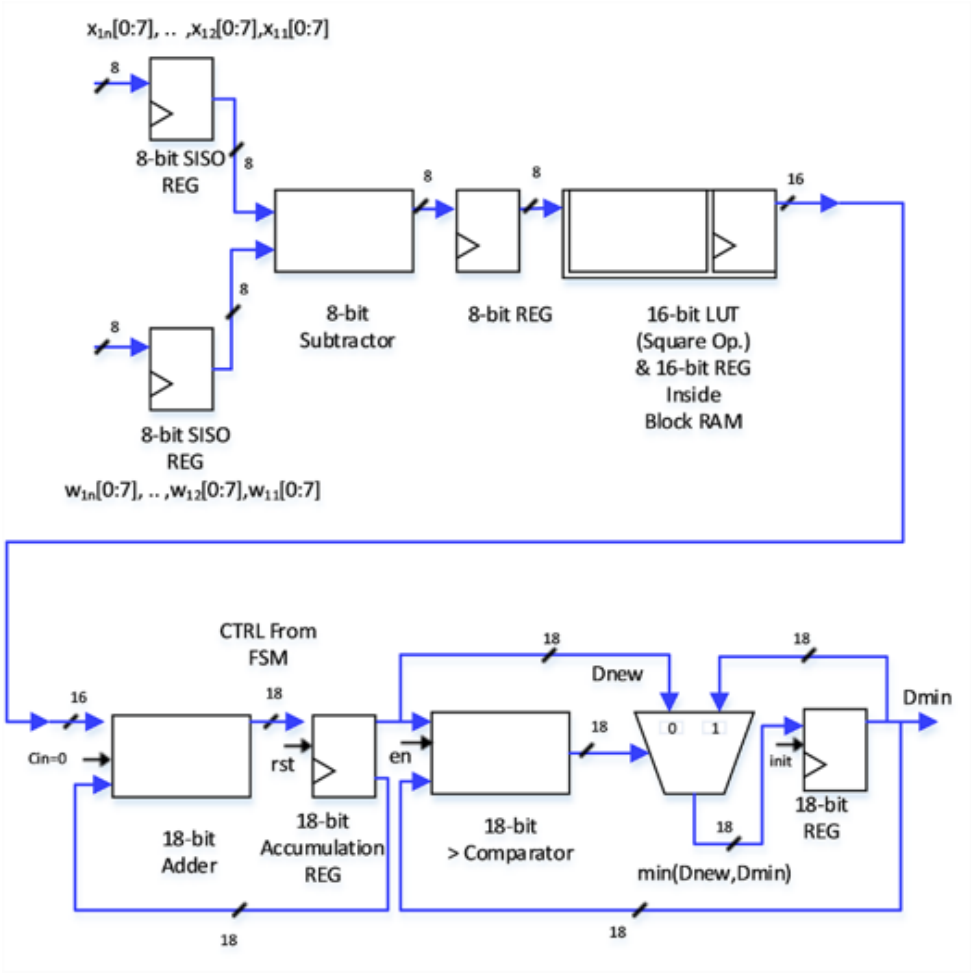


Figure 51: Pipeline Implementation of the Euclidean Distance (ED) Calculator (without square-root function) using standard CMOS FPGA [102].

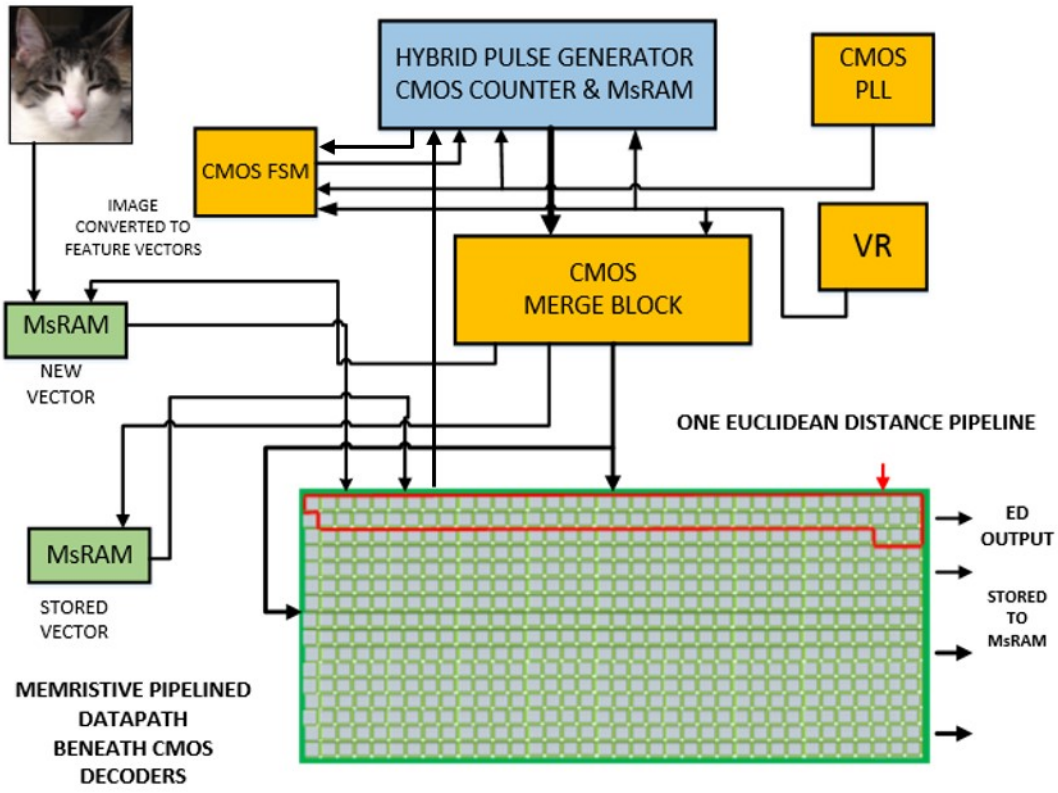


Figure 52: Proposed Memristive stateful logic Field Programmable Gate Array (MsFPGA). The details of the "Hybrid Pulse Generator" and the "CMOS Merge Block" are shown in Figure 17. The red polygon represents one pipeline of the proposed ED architecture and the implementation is illustrated in Figure 53. Color code: Green- memristor nanowire crossbar, Yellow- CMOS, Blue- Hybrid circuitry [102].

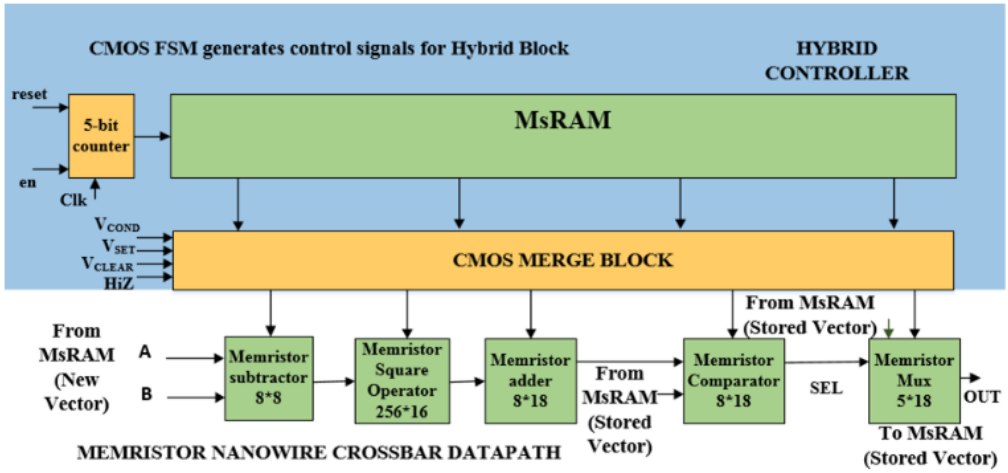


Figure 53: Pipeline Implementation of the Euclidean Distance (ED) Calculator using proposed MsFPGA, memristor-CMOS Hybrid FPGA. Color code: Green-memristor nanowire crossbar, Yellow- CMOS, Blue-hybrid circuitry [103].

A comparative study between CMOS technology and the proposed memristor-CMOS hybrid technology is presented here using the example of the Euclidean Distance calculation pipeline as shown in Figure 51 and 53 respectively [102]. Based on the results presented in Chapter 4 for CMOS FPGA implementation and Chapter 7 for proposed MsFPGA implementation, it is clear that a memristor based technology is a promising alternative for future logic design. The delay numbers calculated in Table 7-2 for the proposed IMPLY-memristor based design using the realistic process based simulation model by Mazadi et al. [17] showed better results compared to the CMOS FPGA based design shown in Table 4-1. Also significant area advantage of the IMPLY-memristor based design was demonstrated, although for the memristor-CMOS hybrid design the CMOS circuitry consumes most of the design area. However, if we exclude the CMOS circuitry of the hybrid design and only compare the area of the ED datapath, we would see a massive

reduction of area (146µm^2 area consumed by IMPLY-memristor nanowire crossbar vs. 0.6mm^2 area consumed by CMOS FPGA). These numbers strongly justify the advantages of the IMPLY-memristor based design over the standard CMOS design.

Besides, in the proposed MsCMOL architecture, protection from the sneak-path current was provided with the proposed methodology in Chapter 6 using an 8-bit iterative adder circuit. This design not only works to minimize the leakage power, but also is protected from flipping bits or logical error. The dynamic power consumed by the example CMOS FPGA at 25% toggle-rate was 22mW, which was much higher compared to the memristor-CMOS hybrid design (~9mW) if both the designs were driven with a supply voltage of 1V. These comparative results are presented in Figure 54, where graphs are plotted in logarithmic scale and the results are also presented in Table 8-1.

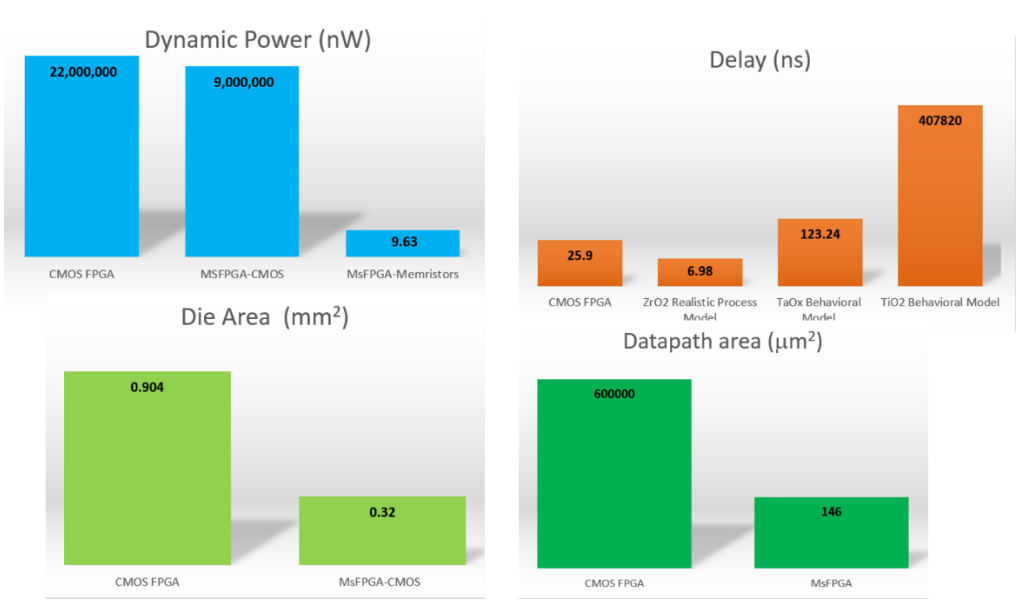


Figure 54: PERFORMANCE COMPARISON OF CMOS FPGA VS. PROPOSED MsFPGA (In Logarithmic scale).

Table 8-1: PERFORMANCE COMPARISON OF CMOS FPGA VS. PROPOSED MsFPGA.

Performances	CMOS FPGA	MsFPGA	CMOS Component of MsFPGA
Delay	25.9ns	ZrO_2 Realistic Process Model: 6.98ns TaO_x Behavioral Model: 123.24ns TiO_2 Behavioral Model: 407.82µs	-
Datapath Area	$0.6mm^2$	$146\mu m^2$	x
Total Die Area	$0.904\ mm^2$	x	$0.32mm^2$
Static Power	Whole FPGA is on, so not comparable.	Sneak-path protection provided.	-
Dynamic Power @25% toggle-rate	22mW	9.63nW	~ 9mW

The pulse frequency for the ZrO_2 memristor variant can be 147GHz [17], which translates into a 6.8ps micro-pulse for every logical operation. I suggest a high-frequency clock to drive the CMOS-memristor hybrid design in the future as it has already been shown that CMOS can be operated at 160GHz frequency or above [23][24][25][26][27][28]. However, running the clock at high frequency would cost significant power dissipation unless provided necessary solutions for that.

9 CONCLUSIONS

This dissertation work has proposed the Memristive stateful logic Field Programmable Gate Array (MsFPGA), a novel and innovative memristor-CMOS hybrid FPGA for digital system design [102]. This architecture is reconfigurable and can be used for many applications, including those that demonstrate massive parallelism [102]. This includes especially various types of neural architectures. In this disserations we are assuming that massive parallel arithmetic operations are possible, as is pipelining. However, it should be obvious to the reader that advantages of the proposed regular design are also applicable to Single Instruction Multiple Data (SIMD)-like, systolic, and CMOL-like datapath-memory architectures that are typical of DSP, neural network and image processing. The proposed MsFPGA is particularly suited to regular designs with rectangular or square blocks executed in parallel. Since the blocks communicate mostly by abutting, the routing is simplified. This makes this architecture particularly well-suited for regular SIMD-like and pipelined architectures. However, because a logical block can also be used for interconnect, in principle the fabric of the MsFPGA can be used for general purpose combinational and sequential functions as presented in [9][10][13]. This dissertation also showed how to eliminate logically dangerous sneak-path current in the nanowire crossbar design using this methodology [102]. The high level architecture, the Memristive Finite State Machine with Datapath (MsFSMD), (which is designed with a CMOL-like datapath-memory, MsCMOL and a memristor-CMOS hybrid controller) was introduced [102]. The hybrid controller has a Pulse-Generation unit, which is based on

Memristive stateful RAM, MsRAM with CMOS interfaces and a small CMOS FSM [102]. This work also proposed a new architecture for calculating Euclidean Distance as a pipelined design, and implemented the hardware with memristors based on implication logic [102]. This dissertation showed a comparison of circuit performance between the proposed memristor-CMOS hybrid design and a pure CMOS FPGA design that shows the significant promise of memristors to be a viable new circuit technology for both memory and combinational logic (including arithmetic) operations [102].

There are a variety of possible research topics that will result from the work presented here. One example involves testing the proposed MsFPGA. This topic is not addressed, with the exception of [46]. A testing method can be proposed that will be similar to the testing of the classical EPLDs, GALs and FPGAs [47][48]. Moreover, this testing method is intended to be used in fault tolerant design, which is able to self-repair using the spare column method in the crossbar.

Contributions

[1] This dissertation has presented a hardware design methodology that is suitable for massively parallel and pipelined reconfigurable architecture. Also, this work implemented the design using the proposed methodology with the IMPLY-memristor based nanowire crossbar [102]. The application areas of the proposed design methodology are, due to the kind of highly parallel, pipelined execution, the methodology enables pattern recognition, robot motion, neural network, big data analysis etc. These application areas include biologically inspired associative memory based models and other similar algorithms.

[2] Using the proposed *space-time based notation* and proposed *pulse generator*, this dissertation presented optimized design for logic blocks using IMPLY-memristors [102]. The list includes critical circuits, such as, XOR (exclusive OR) gate, Half/Full adder, Subtractor, Multiplexer, Comparator design.

[3] In this dissertation, an innovative concept of an 8-bit iterative adder design using the IMPLY-memristor is presented. The 8-bit iterative adder is designed in a new type of 8x8 nanowire crossbar, where, each adder bit is implemented in one row of the 8-row crossbar network [102]. The design is optimized for area and delay and has sneak-path protection [102]. Similarly, components of 8-bit, 16-bit, or any other order bit can be designed using one or multiple 8x8 crossbar blocks, as needed. For this dissertation other arithmetic blocks, e.g. subtractor, comparator, multiplexer, square-operator blocks were also designed using the same design concepts.

[4] The innovative ***pipelining*** concept is presented for the datapath design using an array of 8x8 nanowire crossbar blocks [102]. This array of blocks can grow both horizontally as well as vertically and can act as a pipeline.

[5] A novel Hybrid memristor-CMOS ***MsFPGA (Memristive stateful logic Field Programmable Gate Array)*** [102] design was proposed in this dissertation. The proposed MsFPGA is a reconfigurable system that can be designed with pipelined datapaths and massive parallelism. This ***parallelism*** can be designed by driving many such pipelines (mentioned in [4] above) with one controller simultaneously, using the ***SIMD (Single Instruction Multiple Data)*** concept. These are innovative concepts for the memristive FPGA design, presented by this research.

[6] Several novel architectural concepts were developed. The proposed methodology provides a general new architecture model, ***Memristive stateful Finite State Machine with Datapath (MsFSMD)*** [102]. Like conventional *FSMD*, this proposed system is also a digital system that includes a *finite-state machine*, and a *datapath*, but all logic is stateful and is implemented with memristors, which changes timing and design methods used. Besides, the MsFSMD model has an additional control block called the ***pulse generator*** [102]. The pulse generator can be defined as the brain of the proposed MsFPGA. The pulse generation block contains the ***Memristive stateful RAM (MsRAM)***. The usage of the MsRAM [102], another innovation of this dissertation work, which contains all the configuration information required to realize the virtual logic circuit in the memristive nanowire crossbar datapath.

[7] The proposed MsFPGA uses memristors for memory, connections programming, and combinational logic implementation as opposed to other published memristor based FPGAs, such as mrFPGA, where memristors are reconfigured for logic connections only.

[8] This dissertation proposed solutions to several critical circuit implementation challenges for memristor-nanowire crossbar designs. The proposed ***MsCMOL***, usage of ***data storage MsRAM***, usage of an array of ***8x8 nanowire crossbar blocks,*** the proposed ***sneak-path protection***, and the proposed ***row-to-row data transfer*** are all novel ideas and are valuable to the development of memristor technology [102].

[9] Sneak-path current causes both logical error as well as power consumption in various types of nanowire crossbar designs, including memristor-nanowire crossbar design. This research proposes a design methodology for an innovative, novel sneak-path protected IMPLY-memristive-nanowire crossbar circuit [102]. For this purpose, an example of an 8-bit Full iterative adder design was presented in detail. This design is free of dangerous logical errors and it was minimized for possible power consumption. The power consumption for this proposed design is reduced to the lowest possible level. This sneak-path free combinatorial circuit design methodology proposed by this research is much more robust than any other published research on similar designs with nanowire crossbars.

[10] This dissertation performed the price-performance analysis of CMOS FPGA versus CMOS-memristive hybrid FPGA (Proposed MsFPGA) designs using the Euclidean Distance pipelined datapath [102]. This is a new contribution

as no other published research has presented performance comparisons between two technologies for complete systems with simulated results.

[11] This dissertation proposed the hardware implementation of the ***Euclidean Distance Calculator*** as an innovative ***pipelined datapath*** and presented this datapath as a CMOS FPGA design as well as a memristive FPGA design [102]. Since Euclidean Distance calculation is used in many neural network and associative memory based software algorithms, the hardware realization of the Euclidean Distance Calculator as a *pipelined datapath* with *memristors* is an important concept. This concept can be used in the hardware realization of neumerous application areas, such as, supervised and unsupervised learning, pattern recognition, neural network, hierarchical clustering, phylogenetic analysis, molecular conformation in bioinformatics, dimensionality reduction in machine learning and statistics, natural language text processing, image processing, medical imaging, data mining, big data analysis, shape matching, pedestrian detection, human tracking, action recognition, robot motion planning, shape simplification, volume representation and smoothing Voronoi Diagrams applied in graphics, and robot path planning [19][36][37][38][39][40][41][42][43][44][45].

Journal Publication:

Rahman, K. C., Hammerstrom, D., Li, Y., Xiong, H., & Perkowski, M. (2016). Methodology and Design of a Massively Parallel Memristive Stateful IMPLY Logic based Reconfigurable Architecture. *Nanotechnology, IEEE Transactions on*, xx. (In Production, accepted on 09-May-2016).

Patent:

Rahman, K. C., Perkowski, M., Hammerstrom, D., & Al-Jafar M. filed Provisional Patent Application No. 61/989,387.

REFERENCES

[1] Chua, L. O. (1971). Memristor-the missing circuit element. *Circuit Theory, IEEE Transactions on*, *18*(5), 507-519.

[2] Strukov, D. B., Snider, G. S., Stewart, D. R., & Williams, R. S. (2008). The missing memristor found. *Nature*, *453*(7191), 80-83.

[3] Borghetti, J., Snider, G. S., Kuekes, P. J., Yang, J. J., Stewart, D. R., & Williams, R. S. (2010). 'Memristive' switches enable 'stateful' logic operations via material implication. *Nature*, *464*(7290), 873-876.

[4] Kuekes, P. (2008, November). Material implication: digital logic with memristors. In *Memristor and memristive systems symposium* (Vol. 21).

[5] Kvatinsky, S., Satat, G., Wald, N., Friedman, E. G., Kolodny, A., & Weiser, U. C. (2014). Memristor-based material implication (IMPLY) logic: design principles and methodologies. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, *22*(10), 2054-2066.

[6] Strukov, D. B., & Likharev, K. K. (2005). CMOL FPGA: a reconfigurable architecture for hybrid digital circuits with two-terminal nanodevices. *Nanotechnology*, *16*(6), 888.

[7] Lehtonen, E., Tissari, J., Poikonen, J., Laiho, M., & Koskinen, L. (2014). A cellular computing architecture for parallel memristive stateful logic. *Microelectronics Journal*, *45*(11), 1438-1449.

[8] Likharev, K. K., & Strukov, D. B. (2005). CMOL: Devices, circuits, and architectures. In *Introducing Molecular Electronics* (pp. 447-477). Springer Berlin Heidelberg.

[9] Kim, K., Shin, S., & Kang, S. (2011). Field programmable stateful logic array. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, *30*(12), 1800-1813.

[10] Kim, K., Shin, S., & Kang, S. (2011, May). Stateful logic pipeline architecture. In *Circuits and Systems (ISCAS), 2011 IEEE International Symposium on* (pp. 2497-2500). IEEE.

[11] Kvatinsky, S., Kolodny, A., Weiser, U. C., & Friedman, E. G. (2011, October). Memristor-based IMPLY logic design procedure. In *Computer Design (ICCD), 2011 IEEE 29th International Conference on* (pp. 142-147). IEEE.

[12] Lehtonen, E., & Laiho, M. (2009, July). Stateful implication logic with memristors. In *Proceedings of the 2009 IEEE/ACM International Symposium on Nanoscale Architectures* (pp. 33-36). IEEE Computer Society.

[13] Cong, J., & Xiao, B. (2011, June). mrFPGA: A novel FPGA architecture with memristor-based reconfiguration. In *Nanoscale Architectures (NANOARCH), 2011 IEEE/ACM International Symposium on* (pp. 1-8). IEEE.

[14] Snider, G. S., & Williams, R. S. (2007). Nano/CMOS architectures using a field-programmable nanowire interconnect. *Nanotechnology*, *18*(3), 035204.

[15] http://www.xilinx.com/ (2015, February 23). "XA Kintex-7 FPGAs Overview". DS182 (v2.13).

[16] Biolek, D., Di Ventra, M., & Pershin, Y. V. (2013). Reliable SPICE simulations of memristors, memcapacitors and meminductors. *arXiv preprint arXiv:1307.2717*.

[17] Mazady, A. (2014). Modeling, Fabrication, and Characterization of Memristors.

[18] Torrezan, A. C., Strachan, J. P., Medeiros-Ribeiro, G., & Williams, R. S. (2011). Sub-nanosecond switching of a tantalum oxide memristor. *Nanotechnology*,*22*(48), 485203.

[19] Furao, S., Ogura, T., & Hasegawa, O. (2007). An enhanced self-organizing incremental neural network for online unsupervised learning. *Neural Networks*, *20*(8), 893-903.

[20] Borghetti, J., Li, Z., Straznicky, J., Li, X., Ohlberg, D. A., Wu, W., ... & Williams, R. S. (2009). A hybrid nanomemristor/transistor logic circuit capable of self-programming. *Proceedings of the National Academy of Sciences*, *106*(6), 1699-1703.

[21] Manem, H., Rajendran, J., & Rose, G. S. (2012). Design considerations for multilevel CMOS/nano memristive memory. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, *8*(1), 6.

[22] International technology roadmap for semiconductors. URL http://www.itrs.net/

[23] Razavi, B., Lee, K. F., & Yan, R. H. (1995). Design of high-speed, low-power frequency dividers and phase-locked loops in deep submicron CMOS. *Solid-State Circuits, IEEE Journal of*, *30*(2), 101-109.

[24] Sun, Y., & Herzel, F. (2006). A fully differential 60 GHz receiver front-end with integrated PLL in SiGe: C BiCMOS. In *2006 European Microwave Integrated Circuits Conference* (pp. 198-201).

[25] Pinel, S., Sarkar, S., Sen, P., Perumana, B., Yeh, D., Dawn, D., & Laskar, J. (2008, February). A 90nm cmos 60ghz radio. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International* (pp. 130-601). IEEE.

[26] Tsai, K. H., & Liu, S. I. (2012). A 104-GHz phase-locked loop using a VCO at second pole frequency. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, *20*(1), 80-88.

[27] Lin, Y., & Kotecki, D. E. (2012, August). A 126.9–132.4 GHz wide-locking low-power frequency-quadrupled phase-locked loop in 130nm SiGe BiCMOS. In*Circuits and Systems (MWSCAS), 2012 IEEE 55th International Midwest Symposium on* (pp. 754-757). IEEE.

[28] Chen, W. Z., Lu, T. Y., Wang, Y. T., Jian, J. T., Yang, Y. H., & Chang, K. T. (2014). A 160-GHz Frequency-Translation Phase-Locked Loop with RSSI Assisted Frequency Acquisition.

[29] http://www.orcad.com/ (2015). "OrCAD 16.6 Lite Demo Software (OrCAD Capture and PSpice)". Cadence Design Systems, Inc.

[30] Xia, Q., Robinett, W., Cumbie, M. W., Banerjee, N., Cardinali, T. J., Yang, J. J., ... & Williams, R. S. (2009). Memristor− CMOS hybrid integrated circuits for reconfigurable logic. *Nano letters*, *9*(10), 3640-3645.

[31] Jo, S. H., Chang, T., Ebong, I., Bhadviya, B. B., Mazumder, P., & Lu, W. (2010). Nanoscale memristor device as synapse in neuromorphic systems. *Nano letters*, *10*(4), 1297-1301.

[32] Kim, K. H., Gaba, S., Wheeler, D., Cruz-Albrecht, J. M., Hussain, T., Srinivasa, N., & Lu, W. (2011). A functional hybrid memristor crossbar-array/CMOS system for data storage and neuromorphic applications. *Nano letters*, *12*(1), 389-395.

[33] Yang, J. J., Pickett, M. D., Li, X., Ohlberg, D. A., Stewart, D. R., & Williams, R. S. (2008). Memristive switching mechanism for metal/oxide/metal nanodevices. *Nature nanotechnology*, *3*(7), 429-433.

[34] Raghuvanshi, A., & Perkowski, M. (2014, November). Logic synthesis and a generalized notation for memristor-realized material implication gates. In *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design* (pp. 470-477). IEEE Press.

[35] Kime, C. R., & Mano, M. M. (2004). Logic and computer design fundamentals.

[36] Breu, H., Gil, J., Kirkpatrick, D., & Werman, M. (1995). Linear time Euclidean distance transform algorithms. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, *17*(5), 529-533.

[37] Liberti, L., Lavor, C., Maculan, N., & Mucherino, A. (2014). Euclidean distance geometry and applications. *SIAM Review*, *56*(1), 3-69.

[38] Parhizkar, R. (2013). *Euclidean distance matrices: Properties, algorithms and applications* (Doctoral dissertation, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE).

[39] Coeurjolly, D., & Vacavant, A. (2012). Separable distance transformation and its applications. In *Digital Geometry Algorithms* (pp. 189-214). Springer Netherlands.

[40] Dokmanić, I., Parhizkar, R., Walther, A., Lu, Y. M., & Vetterli, M. (2013). Acoustic echoes reveal room shape. *Proceedings of the National Academy of Sciences*, *110*(30), 12186-12191.

[41] Ye, Q. Z. (1988, November). The signed Euclidean distance transform and its applications. In *Pattern Recognition, 1988., 9th International Conference on* (pp. 495-499). IEEE.

[42] Chu, D. I., Brown, H. C., & Chu, M. T. (2010). On least squares euclidean distance matrix approximation and completion. *Available at February*, *16*.

[43] Barrett, P. (2006). Euclidean distance: Raw, normalised, and double-scaled coefficients. *Unpublished paper retrieved from http://www. pbmetrix. com/techpapers/Euclidean_Distance. pdf*.

[44] Krislock, N., & Wolkowicz, H. (2012). *Euclidean distance matrices and applications* (pp. 879-914). Springer US.

[45] Dokmanic, I., Parhizkar, R., Ranieri, J., & Vetterli, M. (2015). Euclidean Distance Matrices: A Short Walk Through Theory, Algorithms and Applications. *arXiv preprint arXiv:1502.07541*.

[46] Kannan, S., Karimi, N., Karri, R., & Sinanoglu, O. (2015). Modeling, Detection, and Diagnosis of Faults in Multilevel Memristor Memories.*Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, *34*(5), 822-834.

[47] Lee, C. H., Perkowski, M. A., Hall, D. V., & Jun, D. S. (2000). Self-repairable EPLDs: design, self-repair, and evaluation methodology. In *Evolvable Hardware, 2000. Proceedings. The Second NASA/DoD Workshop on* (pp. 183-193). IEEE.

[48] Lee, C. H., Perkowski, M. A., Hall, D. V., & Jun, D. S (2001). Self-Repairable GALs. Journal of Systems Architecture, 02/2001; 47(2):119-135.

[49] Li, H., Gao, B., Chen, Z., Zhao, Y., Huang, P., Ye, H., ... & Kang, J. (2015). A learnable parallel processing architecture towards unity of memory and computing. *Scientific reports*, *5*.

[50] Austin, J., & Stonham, T. J. (1987). Distributed associative memory for use in scene analysis. *Image and Vision Computing*, *5*(4), 251-260.

[51] Bose, J. (2007). *Engineering a Sequence Machine Through Spiking Neurons Employing Rank-order Codes* (Doctoral dissertation, University of Manchester).

[52] Bose, J., Furber, S. B., & Shapiro, J. L. (2005). An associative memory for the on-line recognition and prediction of temporal sequences. In *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on* (Vol. 2, pp. 1223-1228). IEEE.

[53] Buckingham, J., & Willshaw, D. (1992). Performance characteristics of the associative net. *Network: Computation in Neural Systems*, *3*(4), 407-414.

[54] Carpenter, G. A., & Grossberg, S. (1988). The ART of adaptive pattern recognition by a self-organizing neural network. *Computer*, *21*(3), 77-88.

[55] Elman, J. L. (1990). Finding structure in time. *Cognitive science*, *14*(2), 179-211.

[56] Flynn, M. J., Kanerva, P., & Bhadkamkar, N. (1989). Sparse distributed memory: Principles and operation.

[57] Furao, S., & Hasegawa, O. (2006). An incremental network for on-line unsupervised classification and topology learning. *Neural Networks*, *19*(1), 90-106.

[58] Ghahramani, Z. (2004). Unsupervised learning. In *Advanced lectures on machine learning* (pp. 72-112). Springer Berlin Heidelberg.

[59] Govoreanu, B., Kar, G. S., Chen, Y. Y., Paraschiv, V., Kubicek, S., Fantini, A., ... & Jossart, N. (2011, December). 10× 10nm 2 Hf/HfO x crossbar resistive RAM with excellent performance, reliability and low-energy operation. In *Electron Devices Meeting (IEDM), 2011 IEEE International* (pp. 31-6). IEEE.

[60] Guyonneau, R., VanRullen, R., & Thorpe, S. J. (2005). Neurons tune to the earliest spikes through STDP. *Neural Computation*, *17*(4), 859-879.

[61] Hawkins, J., Ahmad, S., & Dubinsky, D. (2010). Hierarchical temporal memory including HTM cortical learning algorithms. *Techical report, Numenta, Inc, Palto Alto http://www. numenta. com/htmoverview/education/HTM_CorticalLearningAlgorithms. pdf*.

[62] Hebb, D. O. (2005). *The organization of behavior: A neuropsychological theory*. Psychology Press.

[63] Holleman, J., Mishra, A., Diorio, C., & Otis, B. (2008, September). A micro-power neural spike detector and feature extractor in. 13µm CMOS. In*Custom Integrated Circuits Conference, 2008. CICC 2008. IEEE* (pp. 333-336). IEEE.

[64] Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, *79*(8), 2554-2558.

[65] Jordan, M. I. (1997). Serial order: A parallel distributed processing approach. *Advances in psychology*, *121*, 471-495.

[66] Kanerva, P. (1992). Sparse distributed memory and related models.

[67] Kim, Y., Zhang, Y., & Li, P. (2012, September). A digital neuromorphic VLSI architecture with memristor crossbar synaptic array for machine learning. In*SOC Conference (SOCC), 2012 IEEE International* (pp. 328-333). IEEE.

[68] Kosko, B. (1988). Bidirectional associative memories. *Systems, Man and Cybernetics, IEEE Transactions on*, *18*(1), 49-60.

[69] Levenshtein, V. I. (1966, February). Binary codes capable or 'correcting deletions, insertions, and reversals. In *Soviet Physics-Doklady* (Vol. 10, No. 8).

[70] LeCun, Y., Cortes, C., & Burges, C. J. (1998). The MNIST database of handwritten digits.

[71] Likharev, K. K. (2011). CrossNets: Neuromorphic hybrid CMOS/nanoelectronic networks. *Science of Advanced Materials*, *3*(3), 322-331.

[72] Loiselle, S., Rouat, J., Pressnitzer, D., & Thorpe, S. (2005, July). Exploration of rank order coding with spiking neural networks for speech recognition. In*Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on* (Vol. 4, pp. 2076-2080). IEEE.

[73] Palm, G., Schwenker, F., Sommer, F. T., & Strey, A. (1997). Neural associative memories. *Associative processing and processors*, 307-326.

[74] Pierzchala, E., & Perkowski, M. A. (1999). *U.S. Patent No. 5,959,871*. Washington, DC: U.S. Patent and Trademark Office.

[75] Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, *77*(2), 257-286.

[76] Sakurai, N., Hattori, M., & Ito, H. (2002, May). SOM associative memory for temporal sequences. In *Proceedings of the 2002 international joint conference on neural networks* (pp. 950-955).

[77] KP, S. S., Turner, A., Sherly, E., & Austin, J. (2014). Sequential data mining using correlation matrix memory. *arXiv preprint arXiv:1407.2206*.

[78] Sharad, M., Augustine, C., Panagopoulos, G., & Roy, K. (2012, June). Cognitive computing with spin-based neural networks. In *Proceedings of the 49th Annual Design Automation Conference* (pp. 1262-1263). ACM.

[79] Shen, F., Yu, H., Kasai, W., & Hasegawa, O. (2010, July). An associative memory system for incremental learning and temporal sequence. In *Neural Networks (IJCNN), The 2010 International Joint Conference on* (pp. 1-8). IEEE.

[80] Shen, F., Ouyang, Q., Kasai, W., & Hasegawa, O. (2013). A general associative memory based on self-organizing incremental neural network. *Neurocomputing*, *104*, 57-71.

[81] Snider, G., Kuekes, P., Hogg, T., & Williams, R. S. (2005). Nanoelectronic architectures. *Applied Physics A*, *80*(6), 1183-1195.

[82] Yamada, T., Hattori, M., Morisawa, M., & Ito, H. (1999). Sequential learning for associative memory using Kohonen feature map. In *Neural Networks, 1999. IJCNN'99. International Joint Conference on* (Vol. 3, pp. 1920-1923). IEEE.

[83] Yang, J. J., Strukov, D. B., & Stewart, D. R. (2013). Memristive devices for computing. *Nature nanotechnology*, *8*(1), 13-24.

[84] Zaveri, M. S., & Hammerstrom, D. (2011). Performance/price estimates for cortex-scale hardware: a design space exploration. *Neural Networks*, *24*(3), 291-304.

[85] Zaveri, M. S., & Hammerstrom, D. (2010). CMOL/CMOS implementations of bayesian polytree inference: Digital and mixed-signal architectures and performance/price. *Nanotechnology, IEEE Transactions on*, *9*(2), 194-211.

[86] Strukov, D. B., Stewart, D. R., Borghetti, J., Li, X., Pickett, M. D., Medeiros-Ribeiro, G., ... & Xia, Q. (2010, May). Hybrid CMOS/memristor circuits. In*ISCAS* (pp. 1967-1970).

[87] Pershin, Y. V., & Di Ventra, M. (2010). Experimental demonstration of associative memory with memristive neural networks. *Neural Networks*,*23*(7), 881-886.

[88] Snider, G. S. (2007). Self-organized computation with unreliable, memristive nanodevices. *Nanotechnology*, *18*(36), 365202.

[89] Likharev, K., Mayr, A., Muckra, I., & Türel, Ö. (2003). CrossNets: High-Performance Neuromorphic Architectures for CMOL Circuits. *Annals of the New York Academy of Sciences*, *1006*(1), 146-163.

[90] Snider, G., Amerson, R., Gorchetchnikov, A., Mingolla, E., Carter, D., Abdalla, H., ... & Patrick, S. (2011). From synapses to circuitry: Using memristive memory to explore the electronic brain. *Computer*, (2), 21-28.

[91] Coleman, J. N., Chester, E. I., Softley, C. I., & Kadlec, J. (2000). Arithmetic on the European logarithmic microprocessor. *Computers, IEEE Transactions on*, *49*(7), 702-715.

[92] Taylor, F. J., Gill, R., Joseph, J., & Radke, J. (1988). A 20 bit logarithmic number system processor. *Computers, IEEE Transactions on*, *37*(2), 190-200.

[93] Eshraghian, K., Cho, K. R., Kavehei, O., Kang, S. K., Abbott, D., & Kang, S. M. S. (2011). Memristor MOS content addressable memory (MCAM): Hybrid architecture for future high performance search engines. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, *19*(8), 1407-1417.

[94] Lehtonen, E., Poikonen, J. H., & Laiho, M. (2012, August). Applications and limitations of memristive implication logic. In *Cellular Nanoscale Networks and Their Applications (CNNA), 2012 13th International Workshop on* (pp. 1-6). IEEE.

[95] Patterson, D. A., & Hennessy, J. L. (2013). *Computer organization and design: the hardware/software interface*. Newnes.

[96] Hu, X., Duan, S., & Wang, L. (2012, November). A novel chaotic neural network using memristive synapse with applications in associative memory. In *Abstract and Applied Analysis* (Vol. 2012). Hindawi Publishing Corporation.

[97] Chu, P. P. (2006). *RTL hardware design using VHDL: coding for efficiency, portability, and scalability*. John Wiley & Sons.

[98] Barrabés Castillo, A. (2012). Design of single precision float adder (32-bit numbers) according to IEEE 754 standard using VHDL.

[99] Zidan, M. A., Fahmy, H. A. H., Hussain, M. M., & Salama, K. N. (2013). Memristor-based memory: The sneak paths problem and solutions. *Microelectronics Journal*, *44*(2), 176-183.

[100] Lehtonen, E. (2012). Memristive computing. *University of Turku, Finland*.

[101] Shevgoor, M., Muralimanohar, N., Balasubramonian, R., & Jeon, Y. (2015, October). Improving memristor memory with sneak current sharing. In*Computer Design (ICCD), 2015 33rd IEEE International Conference on* (pp. 549-556). IEEE.

[102] Rahman, K. C., Hammerstrom, D., Li, Y., Xiong, H., & Perkowski, M. (2016). Methodology and Design of a Massively Parallel Memristive Stateful IMPLY Logic based Reconfigurable Architecture. *Nanotechnology, IEEE Transactions on*, xx. (In Production, accepted on 09-May-2016).

Appendix A - SOFTWARE – MATLAB CODES FOR ESOINN & GAM (PATTERN RECOGNITION ALGORITHMS)

The motivation of this work is explained in detail in Chapter 1 of this dissertation. For this research, various biologically inspired i.e. neural network based pattern recognition algorithms were studied. In order to understand and compare the performance, some of these promising algorithms were coded in MATLAB as a part of this research. The pseudo codes were published in many papers [51][52][61][73][79][80] and most of those algorithms were coded for this research. These algorithms were then simulated and a comparative performance for the pattern recognition application was conducted. Based on the performance comparison, GAM (General Associative Memory) [80] was found to be the best algorithm for both spatial and temporal pattern recognition.

Although the original goal of this dissertation work was to develop a hardware design methodology for this best performing algorithm for its complete system, however, later we realized that the implementation of the complete system is unnecessary for providing the design methodology. Therefore, a common critical hardware component was selected to develop the design methodology that is used by most of the neural network and machine learning based algorithms. This component is the Euclidean Distance (ED) Processor/Calculator. ED calculator can be used in a massively parallel and pipelined datapath systems and thus it can have applications in pattern recognition, robot motion, big data analysis, image processing, voice recognition, DSP, database mining and may other hardware systems where large number of wide vectors need to be processed simultaneously.

The ESOINN [79] and GAM [80] codes are presented in Appendix A.

ESOINN (Enhanced Self-Organizing Incremental Neural Network) Model:

The ESOINN algorithm is an *Unsupervised Learning* algorithm for Spatial Pattern Recognition. Unsupervised learning [79] studies how a system can learn to represent particular input patterns in a way that reflects the statistical structure of the overall collection of input patterns. By contrast with supervised learning or reinforcement learning, in unsupervised learning there are no explicit target outputs or environmental evaluations associated with each input.

GAM (General Associative Memory) Model: This algorithm is an improved version of ESOINN for temporal pattern recognition.

Handwritten digit database

This training dataset used for the algorithm was derived from the original MNIST database available at http://yann.lecun.com/exdb/mnist/ [70]

The training data file for each class 0 to 9 was generated.

File format:

Each file has 1000 training examples. Each training example is of size 28x28 pixels. The pixels are stored as unsigned chars (1 byte) and take values from 0 to 255. The first 28x28 bytes of the file correspond to the first training example, the next 28x28 bytes correspond to the next example and so on.

Algorithm Organization:

- Many classes

- Many sub-classes under each class
- Many nodes under each sub-class

Data Structure:

Data of 10 classes - 0 – 9

In each class there are 1000 samples

So potentially there are 10*1000 = 10K nodes

Node = 28x28 = 784 elements with values between 0 to 255

Node = A vector of 784 elements with values from 0 to 255

Each image is node -> sub-class -> class

When an image is received, first its class is found and then its subclass is identified.

Class will be indexed/identified by numbers 0 - variable

Sub-class will be indexed/identified by numbers 0 - variable

Nodes will be indexed/identified by numbers 0 - variable

Image can be catalogued --> NODE[CI][SCI][NI]

CI -> Class index

SCI -> Sub-class index

NI -> Node index

Node has a local maximum density -> apex of a sub class

Image Database: The image database is populated with 10,000 images, of which, the node distribution for digits 0 through 9 is shown in Figure A-1. Each digit between 0 through 9 represents a *Class*. Each category in Figure A-1 has 1000 images and each of these images represents a *Subclass* under the *Class*. Figure A-2 shows MNIST handwritten digits. Each digit has a pixel size of 28x28. The pixels are stored as unsigned chars (1 byte) and take gray-scale values from 0 to 255. The first 28x28 bytes of the training file correspond to the first training example, the next 28x28 bytes correspond to the next example and so on. As such, 10,000 lines were concatenated in one training file.

Nodes	Represent Image for Digit
1-1000	0
1001-2000	1
2001-3000	2
3001-4000	3
4001-5000	4
5001-6000	5
6001-7000	6
7001-8000	7
8001-9000	8
9001-10000	9

Figure A- 1: Node Representation for Various Digits.

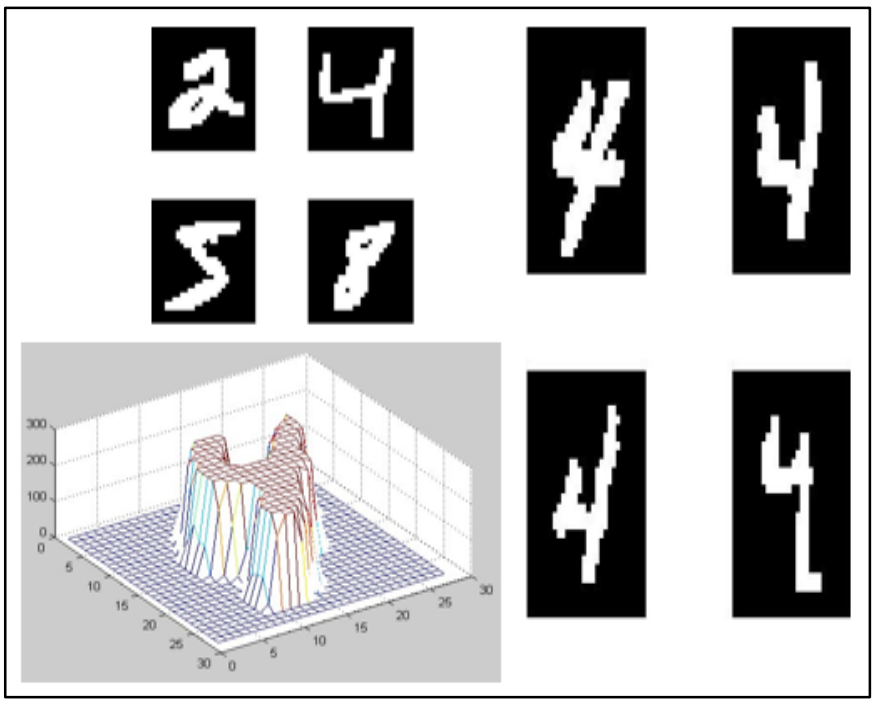


Figure A- 2: MNIST Handwritten Digits used in the experiments. Upper left: Classes 2, 4, 5, 8; Right: Subclasses of digit 4; Lower left: 3-D image of digit 4.

Training database: There are 400 images randomly picked from the image database and used for training, of which nodes 1-100 represent image 2; nodes 101-200 represent image 4; nodes 201-300 represent image 5 and nodes 301-400 represent image 8.

Test database: There are 200 images randomly picked from the image database and used for testing, of which nodes 1-50 represent image 2; nodes 51-100 represent image 4; nodes 101-150 represent image 5 and nodes 151-200 represent image 8. The number of test images is a smaller set compared to the number of training images.

Distance Threshold constant:

A distance threshold constant is used to control the classification of a new node to a new class or to an existing class. During the experimentation, the values of distance threshold are changed several times. A small value of distance threshold may result in a large number of classes. For example, after some trial and error, for the four broader input classes (digits 2, 4, 5, 8) as mentioned above, a large number of classes can be obtained at the output. With further experimentation, it is possible to obtain even fewer classes at the output by iterating on the distance threshold constant.

ESOINN MODEL:

readdata.m

```
clear all
%open the file corresponding to digit
k=1;
l=1;
for j=[1 4 5 8]
    filename = strcat('MNIST\data',num2str(j),'.txt');
    [fid(k) msg] = fopen(filename,'r');
    filename
    %read in the first training example and store it in a 28x28 size
matrix t1
    for i=1:100
        [data28x28,N]=fread(fid(k),[28 28],'uchar');
        data(l,:) = reshape(data28x28,1,28*28);
        dataX = reshape(data28x28,1,28*28);
        l = l+1;
        %imshow(data28x28');
        %pause(0.5)
    end
    k = k+1;
end
save ('numimagedat4_1.mat','data');
```

distcalc.m

```
function z = distcalc(w,p)

%DIST Euclidean distance weight function.
%  Algorithm
%  The Euclidean distance D between two vectors X and Y is:
%  D = sqrt(sum((x-y).^2))

[S,R] = size(w);
[R2,Q] = size(p);
if (R ~= R2), error('Inner matrix dimensions do not match.'),end

z = zeros(S,Q);
if (Q<S)
  p = p';
  copies = zeros(1,S);
  for q=1:Q
    z(:,q) = sum((w-p(q+copies,:)).^2,2);
  end
else
  w = w';
  copies = zeros(1,Q);
  for i=1:S
    z(i,:) = sum((w(:,i+copies)-p).^2,1);
  end
end
```

```
z = sqrt(z)/R;
```

findthreshold.m

```
% given a set of nodes, find maximum & minimum sim_threshold of each of
the nodes.
function [TMax, TMin] = findthreshold(a,DIST_THRESH)

[NRow,MCol] = size(a);

for i=1:NRow % assuming I have 100 nodes

    TMax(i) = 0;
    TMin(i) = 9999;

    for j=1:NRow
        dist =  distcalc (a(i,:), a(j,:)');
        %fprintf('%f %f\n',DIST_THRESH, dist);
        if(dist < DIST_THRESH)
            if dist > TMax(i)
                TMax(i) = dist;
            end
            if dist < TMin(i)
                TMin(i) = dist;
            end
        end

    end

end

return
```

findwinners.m

```
% given a set of nodes, find winner and second winner.
function [winner, winner2, DWinner, DWinner2] = findwinners(a,x)

[NRow,MCol] = size(a);

for i=1:NRow % assuming I have 100 nodes
    dist(i) =  distcalc (x, a(i,:)');
end

if dist(1) < dist(2)
    winner = 1;
    winner2 = 2;
else
    winner = 2;
    winner2= 1;
end

for i= 3:NRow
```

```
    if dist(i) < dist(winner)
        temp = winner;
        winner = i;
        if dist(winner2) > dist(temp);
           winner2 = temp;
        end
    else
        if dist(i) < dist(winner2)
            winner2 = i;
        end
    end
end
DWinner = dist(winner);
DWinner2 = dist(winner2);
return
```

find winnersX.m

```
% given a set of nodes, find winner and second winner.
function [winner, winner2, DWinner, DWinner2] = findwinnersX(a,x)

[NRow,MCol] = size(a);

for i=1:NRow % assuming I have 100 nodes
    dist(i) =  distcalc (x, a(i,:)');
end

if dist(1) < dist(2)
    winner = 1;
    winner2 = 2;
else
    winner = 2;
    winner2= 1;
end

for i= 3:NRow

    if dist(i) < dist(winner)
        temp = winner;
        winner = i;
        if dist(winner2) > dist(temp);
            winner2 = temp;
        end
    else
        if dist(i) < dist(winner2)
            winner2 = i;
        end
    end
end
DWinner = dist(winner);
DWinner2 = dist(winner2);
% if DWinner == 0
%     DWinner
```

```
%     sparse(a)
%     sparse(x)
% end

Return
```

find_neighbors.m

```
function [nghbrs] = find_neighbors(winner, W, DIST_THRESH)

% find how many nodes in the sub space

[SR SC] = size( W);
cnt = 1;
for i=1:SR
    dist = distcalc(W(winner,:), W(i,:)');
    if(dist < DIST_THRESH)
        nghbrs(cnt) = i;
        cnt = cnt + 1;
    end
end


end

return
```

soinn_subclass.m

```
clear all
load soinn_400.mat

% pick class

[RC SC] = size(class_of_node);

% initialize

for i = 1:SC
    visited(i) = 0;
    subclass(i) = 0;
end

% now do the classification

% "Connections matrix" is tracking all the connected nodes of a given
node

for i = 1:SC
    k = 1;
    for j = 1:SC
        if(i ~= j)
            if (Conn(i,j) == 1)
```

```
                Connections(i,k) = j; % Connection recorded
                k = k + 1;

            end
        end
    end
end



% Find density of each node
for p = 1:NClass
    scindx = 1;

    for i = 1:SC
        if ((visited(i) == 0) && (class_of_node(i) == p))
            k=1;
            clear visited_t;
            %fprintf ('class = %d node = %d\n',p,i);
            marker = 99;
            max = h(i);
            max_node = i;

            visited_t(k) = i; % Keepingtrack of visited tree
            visited(i) = 1;   % Keeping track of the nodes that are
already worked on

            current_node = i;
            new_marker = marker + 1 ; % this is a way to flag the last
node of the tree

            [max, max_node, new_marker, visited, visited_t, k] =
search_node_tree(Connections, max, max_node, marker, current_node, k,
h, visited_t, visited);

            while (new_marker > marker)
                marker = new_marker;
                [max, max_node, new_marker, visited, visited_t, k] =
search_node_tree(Connections, max, max_node, marker, current_node, k,
h, visited_t, visited);
                current_node = max_node;
            end
            % done searching that tree
            % assign sub-class here
            [X, TNodesInTree] = size(visited_t);
            %disp ('visited_tree')
            visited_t;
            %disp('visited of current node')
            visited(current_node);

            for m=1:TNodesInTree
               subclass(visited_t(m) ) = scindx;
            end
```

```
            subclass_elems{p,scindx,:} = visited_t;
            subclass_apex{p,scindx} = max_node; % Node with highest
density of a given subclass
            scindx = scindx + 1;
        end

    end
    p;
    scindxcount(p) = scindx -1;

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  For testing writing the results to a text file
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fileID = fopen('organize.txt','w');
for i = 1: SC
%     %fprintf (fileID, 'class = %d subclass = %d node =
%d\n',class_of_node(i), subclass(i), i);
     fprintf (fileID, '%d %d %d\n',class_of_node(i), subclass(i), i);
 end
 fclose(fileID);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Following is needed for subclass merging
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

for p = 1:NClass
    for m = 1:scindxcount(p)
        sum(p,m) = 0;
        count(p,m) = 0;
    end
end

for p=1:NClass
    for m=1:scindxcount(p)
        for i=1:SC
            if( (class_of_node(i) == p) && (subclass(i) == m))
                sum(p,m) = sum(p,m) + h(i);
                count(p,m) = count(p,m) + 1;
            end
        end
    end
end

for p=1:NClass
    for m=1:scindxcount(p)
        Avrg(p,m) = sum(p,m)/count(p,m);
    end
end

[dataR dataC] = size(W);
```

```
for p=1:NClass
    fprintf('Total elements in class %d is %d\n',p,scindxcount(p));
    for m=1:scindxcount(p)
        clear other_nodes;
        if(scindxcount(p) > 1) % there is no point of finding winner
and second-winners to other subclasses when we have only 1 subclass
            mxnode = subclass_apex{p,m};

            for j=1:scindxcount(p)
                scwinner(p,m,j) = 0;
                scwinner2(p,m,j) = 0;
                scDWinner(p,m,j) = 0;
                scDWinner2(p,m,j) = 0;
                all_elems_of_subclass = subclass_elems{p,j,:};
                [A Sz] = size(all_elems_of_subclass);
                other_nodes = zeros(Sz,dataC);

                for i=1:Sz
                    other_nodes(i,:) = W(all_elems_of_subclass(i),:);
                end
                subclass_elems{p,j,:}
                if(Sz == 1)
                    SnglNode = subclass_elems{p,j,:};
                    scwinner(p,m,j) = subclass_elems{p,j,:};
                    scwinner2(p,m,j) = subclass_elems{p,j,:};
                    scDWinner(p,m,j) = distcalc(W(SnglNode,:),
W(mxnode,:)');
                    scDWinner2(p,m,j) = scDWinner(p,m,j);
                else
                    MoreNodeArray = subclass_elems{p,j,:};
                    [WW1,WW2,scDWinner(p,m,j), scDWinner2(p,m,j)] =
findwinnersX(other_nodes,W(mxnode,:));
                    scwinner(p,m,j) = MoreNodeArray(WW1);
                    scwinner2(p,m,j) = MoreNodeArray(WW2);
                end
                clear other_nodes;
                fprintf ('p=%d m=%d, winner=%d,
winner2=%d\n',p,m,scwinner(p,m,j), scwinner2(p,m,j));
            end
        end

    end
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Check if the two subclasses need to be merged
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for p=1:NClass
    for m=1:scindxcount(p)
        for j=1:scindxcount(p)
            fprintf ('==>[%d %d %d] %d %d %f
%f\n',p,m,j,scwinner(p,m,j), scwinner2(p,m,j),scDWinner(p,m,j),
scDWinner2(p,m,j));
```

```
        end
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% If nodes from two sub classes are connected -> disconnect
% This is true for even if the two subclasses belong to two different
class
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

for i = 1:SC
    for j = 1:SC
        if((i ~= j) && (subclass(i) ~= subclass(j)))
            if (Conn(i,j) == 1)
                Conn(i,j) = 0;
            end
        end
    end
end
```

subclass_test.m

```
load soinn.mat
[SA SB] = size(class_of_node);

for ii = 1:SB
    if(class_of_node (ii) == 2)
        point_density(ii)
    end
end
```

updt_winner.m

```
function  [A] = updt_winner(winner, x, W, M)

[SR SC] = size( W);

for j = 1:SC
    dW(j) = x(j) - W(winner,j);
    A(j) = W(winner,j) + dW(j)/M(winner);
end

return
```

updt_neighbors.m

```
function  [W] = updt_neighbors(winner, nghbrs, x, W, M)

[SR SC] = size( W);
```

```
[SNR SNC] = size(nghbrs);

for k = 1: SNC
    if(nghbrs(k) ~= winner) % We do not want to update winner again
        for j = 1:SC
            dW(j) = x(j) - W(nghbrs(k),j);
            W(nghbrs(k)) = W(nghbrs(k),j) + dW(j)/(100*M(winner));
            %fprintf('neighbor node = %d\n',nghbrs(k));
        end
    end

end

return
```

updt_connection_matrix.m

```
function [Conn] = update_connection_matrix (Conn, CN, value)

[SR, SC] = size(Conn);

for i = 1:SR
    Conn(CN,SR) = value;
end

return;
```

updt_conn_edge_n_point_density.m

```
function [Conn, Age, point_density] =
update_conn_edge_n_point_density(W, Conn, Age, winner)

% Conn -- Connectivity matrix
% W -- Weight vectors of each node
% Age -- age of each connection. So all possible connection edge will
have
% an "age" value
% winner - winner node
% Size of connection matrix will determine the
% size of existing node space
%disp('Weight::')
%W

[SR, SC] = size(Conn);

Agemax = 100;


point_density = zeros(SR,1);
avg_density = zeros(SR,1);

% Search for all connectivity to winner and update their connection age
```

```
for i = 1: SC
   if Conn(winner, i) == 1
        Age(winner, i) = Age(winner, i) + 1;

        if Age(winner, i) > Agemax
            Conn(winner, i) = 0;
        end
    end
end

% Now calculate the point density of ALL the nodes
for i = 1: SR
    dist = 0;
    M=0; % Number of connections with the given node "i"
    for j = 1: SC
        if i ~= j
            if Conn(i, j) == 1
                % W(i,:)
                % W(j,:)
                dist = dist + distcalc(W(i,:),W(j,:)');
                M = M + 1;
            end
        end
    end

    % Calculate Average Density
    if(M > 0)
        avg_density(i) = dist/M;
    else
       avg_density(i) = 0;
    end
    if M == 0
        point_density(i) = 0;
    else
        point_density(i) = 1/ (1 + avg_density(i))^2;
    end
end
return
```

search_node_tree.m

```
function [max, max_node, new_marker, visited, visited_t, k] =
search_node_tree (Connections, max, max_node, marker, current_node, k,
h, visited_t, visited)

% Now lets find the largest connected tree because that will determine
the
% final size of the "Connections" matrix

[CR, CC] = size(Connections);
new_marker = marker;
k;

for jc = 1:CC % checking connections of the nodes connected to i
```

```
    j = Connections(current_node,jc);
    if ( j ~= 0)
        %fprintf ('=> %d %d %d %d %d\n',current_node, j, k,
visited(current_node), visited(j));
    end
    if (  ( j ~= 0) && (max_node ~= j))
        if (visited(j) ~= 1)
            k = k+1;
            visited_t(k) = j;
            visited(j) = 1;
            if (h(j) > h(max_node))
                max = h(j);
                max_node = j;
                new_marker = marker + 1;
            end
        end
        %fprintf ('===> %d %d %d\n',current_node, j, k);
        visited_t;
    end
```

soinn.m

```
clear all
load numimagedat4.mat

% Select two random entries from the image database to
% initialize the SOINN system
dist = 0;

[DataSize,DataElems] = size(data);

DIST_THRESH = 3.00; %% used for determining the neighboring nodes

while(dist < 2.5)
    randindx1 = (round(rand(1)*(DataSize-1)) +1);
    randindx2 = (round(rand(1)*(DataSize-1)) +1);
    W(1,:) = data(randindx1,:);
    W(2,:) = data(randindx2,:);
    sd = 0;

% i stands for row vector and ik stands for column values in each row

    for ik=1:784
        sd = sd + (W(1,ik) - W(2,ik))^2;
    end
    dist = sqrt(sd)/784;
    TMax(1) = dist;
    TMax(2) = dist;
end
% Now the system has two nodes
N= 2;
NClass = 2;
%class(class,node#)=node#
```

```
class_of_node(1) = 1;
class_of_node(2) = 2;

Conn(1,1) = 1;
Conn(1,2) = 0;
Conn(2,1) = 0;
Conn(2,2) = 1;

Age(1,1) = 0;
Age(1,2) = 0;

M(1) = 1;
M(2) = 1;

% Introduce new nodes (i.e. images) to the system
for i = 1: DataSize-2
    indx = i;
    % CN --- index of the nodes as a new input is introduced
    CN = 2 + i;
    x = data(indx, :);
    Conn(CN,CN) = 1;
    Age(CN,CN) = 0;
    [winner, winner2,DWinner, DWinner2] = findwinners(W,x);
    W(CN,:)= x;
    M(CN) = 1;
    % update connection matrix for the new member with no connection to
    % begin with
    [Conn] = update_connection_matrix (Conn, CN, 0);
    % W - Weight matrix
    % Conn - Connection matrix
    % Age = Age matrix
    % winner - ID of the winner node
    if DWinner > TMax(winner) % A new class.
        NClass = NClass+1;
        class_of_node(CN) = NClass;
        [TMax, TMin] = findthreshold(W,DIST_THRESH);
        Conn(CN, winner) = 0;
        Age(CN, winner) = 0;
        Conn(CN, winner2) = 0;
        Age(CN, winner2) = 0;
        point_density(CN) = 0;
        size(Conn);

    else % step4 - member of existing class of the winner node
        class_of_node(CN) = class_of_node(winner);
        M(winner) = M(winner) + 1;
        [TMax, TMin] = findthreshold(W,DIST_THRESH);
        Conn(CN, winner) = 1; % establishing a connection between
winner and the new node
        Conn(winner, CN) = 1;
        dw1w2 = distcalc(winner, winner2);
        Age(CN, winner) = 0; % setting age to 0
        Age(winner, CN) = 0;
        if(dw1w2 < DIST_THRESH)
```

```
            Conn(winner, winner2) = 1;
            Conn(winner2, winner) = 1;
            Age(winner, winner2) = 0;
            Age(winner2, winner) = 0;
        end
        %%% Update weight of winner and its neighbors
        % find neighbors of winner
        [nghbrs] = find_neighbors(winner, W, DIST_THRESH);
        % update weight of winner
        [W(winner,:)] = updt_winner(winner, x, W, M);
        % update weight of neighbor
        [W] = updt_neighbors(winner, nghbrs, x, W, M);
        % disp('Weight::');
        %W
        [Conn, Age, point_density] =
update_conn_edge_n_point_density(W, Conn, Age, winner);
        % Now that I updated the point density of one node, I need to
        % update the accumulated point density of every one

    end
    size(point_density);
    point_density';
    for kk = 1: i-1

       % kk is the row and CN is the column.
       % kk tracks the history of the
       % previous learnings as a row of the
       % "point_density_history" matrix.
       % Since each row has to hold same number
       % of columns and as we learn
       % new items, number of columns grow,
       % we have to zero pad the earlier
       % rows to accommodate the size growth for the new entry

       point_density_history(kk,CN) = 0;
    end

    point_density_history(i,:) = point_density';
    [sr, sc] = size(point_density_history);

    for col = 1:sc
        NN = sum(spones(point_density_history(:,col)));
        accum_point_density(col) = sum(point_density_history(:,col));
        mean_accum_point_density(col) = accum_point_density(col)/NN;
        h(col)= mean_accum_point_density(col);
    end

end

save('soinn_400.mat')
```

GAM MODEL:

- soinn_12_train_v0: Implementation of algorithm 1 & 2 for training the memory layer and creating the associative layer.

```
% In algorithm at first we put all nodes into one class
% For training you go with known classes of data as suggested in GAM
% Or you go with unsupervised learning as suggested in SOINN
%
% ALGORITHM 1: Learning of the memory layer
% ALGORITHM 2: Building Associative Layer

clear all
tic
for ClsName=1:10
    FName = strcat('traindata_p',num2str(ClsName),'.mat');
    FName
    load ( FName );
    [DataSize,DataElems] = size(data);

    % introduce new node - Step 4
    Class(ClsName).Node(1).W = data(1,:);
    Class(ClsName).Node(1).Th = 0;
    Class(ClsName).Node(1).M = 1; % Frequency of winning of that node
    Class(ClsName).Node(1).N = 0; %
    Class(ClsName).NodeCount = 1;
    ClassCount = 1;
    Class(ClsName).ConnMatrix(1,1) = 1;
    Class(ClsName).ConnAge(1,1) = 1;


    for indx = 2: DataSize
        x = data(indx,:);
        DoneClassification = 0; % Reset it every time
                                % you processed a new node
        XX= ['Training Class => ',num2str(ClsName),' New data =>
',num2str(indx)];
        disp(XX);

        % Find winner and second winner - step 6 - 8

        WinnerNode = 1;
        Winner2Node = 1;
        WinnerDistance = 0;
        Winner2Distance = 0;

        for Ni = 1:Class(ClsName).NodeCount
            dist = distcalcSOINON(Class(ClsName).Node(Ni).W ,x);
```

```
            %dd = sprintf ('Now Processing indx:  %5d -> Node: %5d
dist: %f [Node Th: %f]' , indx, Ni, dist, Class(ClsName).Node(Ni).Th );
            %disp(dd);

            if (dist > Class(ClsName).Node(Ni).Th)      % Step 8
                %disp('dist > thr');
                if Class(ClsName).Node(Ni).Th == 0
                    %disp('=> Wd = 0');
                    WinnerNode = Ni;
                    Winner2Node = Ni;
                    WinnerDistance = dist;
                    Winner2Distance = dist;
                    Class(ClsName).Node(Ni).Th = dist;
                else
                    if WinnerDistance == Winner2Distance
                        %disp( '=> Wd == W2d');
                        if WinnerDistance == 0
                            Winner2Node = Ni;
                            Winner2Distance = dist;
                            WinnerNode = Ni;
                            WinnerDistance = dist;
                        elseif dist > WinnerDistance
                            Winner2Node = Ni;
                            Winner2Distance = dist;
                        else
                            WinnerNode = Ni;
                            WinnerDistance = dist;
                        end
                    elseif dist < Winner2Distance
                        %disp('=> dist < W2d');
                        Winner2Node = Ni;
                        Winner2Distance = dist;
                    else
                        %disp(['  > th but
..',WinnerDistance,Winner2Distance]);
                    end
                end

            else
                % Update winner and second winner - Step 6
                if dist <=  Class(ClsName).Node(Ni).Th
                    Winner2Distance = WinnerDistance;
                    Winner2Node = WinnerNode;
                    WinnerDistance = dist;
                    WinnerNode = Ni;

                elseif dist < Winner2Distance

                    Winner2Distance = dist;
                    Winner2Node = Ni;
                end
            end
```

```
            %dd = sprintf ('Node:  %5d -> Wd: %5.3f WN: %5d W2d: %5.3f
W2N: %5d' , Ni, WinnerDistance,WinnerNode, Winner2Distance, Winner2Node
);
            %disp(dd);
        end

        %Class(Ci).NodeCount


        %dd = sprintf('Classification Done for indx: %d, NodeCount: %d,
Wd: %f Th: %f', indx,Class(ClsName).NodeCount,WinnerDistance,
Class(ClsName).Node(WinnerNode).Th);
        %disp(dd);
        Class(ClsName).Node(WinnerNode).M =
Class(ClsName).Node(WinnerNode).M + 1; % step 6

        if WinnerDistance > Class(ClsName).Node(WinnerNode).Th % Step 8
            %disp( ['introduce new node to the class', WinnerDistance,
' > ' ,Class(ClsName).Node(WinnerNode).Th ]);
            NNi = Class(ClsName).NodeCount+1;
            Class(ClsName).NodeCount = Class(ClsName).NodeCount + 1;
            Class(ClsName).Node(NNi).W = x;
            Class(ClsName).Node(NNi).M = 1;
            Class(ClsName).Node(NNi).N = 0;
            % Update thresholds

            Class(ClsName).Node(NNi).Th = dist;
            Class(ClsName).Node(Ni).Th = dist;
        elseif WinnerDistance  == Winner2Distance
            %disp( ['introduce new node to the class', WinnerDistance '
== ' ,Winner2Distance]);
            NNi = Class(ClsName).NodeCount+1;
            Class(ClsName).NodeCount = Class(ClsName).NodeCount + 1;
            Class(ClsName).Node(NNi).W = x;
            Class(ClsName).Node(NNi).M = 1;
            Class(ClsName).Node(NNi).N = 0;
            % Update thresholds
            Class(ClsName).Node(NNi).Th = dist;
            Class(ClsName).Node(Ni).Th = dist;
        else  % Step 10
            delS1 = 1/Class(ClsName).Node(WinnerNode).M;
            delS2 = 1/Class(ClsName).Node(Winner2Node).M;
            Class(ClsName).Node(WinnerNode).W =
Class(ClsName).Node(WinnerNode).W + delS1*(x-
Class(ClsName).Node(WinnerNode).W); % eq 10
            Class(ClsName).Node(Winner2Node).W =
Class(ClsName).Node(Winner2Node).W + delS2*(x-
Class(ClsName).Node(Winner2Node).W); % eq 11
            Class(ClsName).Node(WinnerNode).Th =
(Class(ClsName).Node(WinnerNode).Th + WinnerDistance)/2; %eq 12
        end
        Class(ClsName).ConnMatrix(WinnerNode,Winner2Node) = 1; %Step 13
        Class(ClsName).ConnAge(WinnerNode,Winner2Node) = 0; %Step 14
        Class(ClsName).ConnMatrix(Winner2Node,WinnerNode) = 1; %Step 13
```

```
        Class(ClsName).ConnAge(Winner2Node,WinnerNode) = 0; %Step 14
        %image(reshape((Class(ClsName).Node(WinnerNode).W),28,28)')
        %pause(1)
        % Step 15
        [NS_1 NS_2] = size(Class(ClsName).ConnAge(WinnerNode,:));
        for jk = 1:NS_2
            if Class(ClsName).ConnMatrix(WinnerNode,jk) == 1
                Class(ClsName).ConnAge(WinnerNode,jk) =
Class(ClsName).ConnAge(WinnerNode,jk) + 1;
            end
        end
    end

    [Ns1 Ns2] = size(Class(ClsName).Node);

    MostVisNode = 1;
    MostVisNodeM = 1;
    for Mn=1:Ns2
        if Class(ClsName).Node(Mn).M > MostVisNodeM
            MostVisNode = Mn;
            MostVisNodeM = Class(ClsName).Node(Mn).M;
        end
    end

    % Build associative layer
    AssocClass(ClsName).Wb = Class(ClsName).Node(MostVisNode);
    AssocClass(ClsName).Mb = 0;


end



save('soinn_trained_assoc.mat')
toc
```

- soinn_2_v0: training the associative layer with temporal sequence.

```
% Learning of the associative layer
% 2-4-1-3
% key-rwaponse vector
% 2-4
% 4-1
% 1-3
clear all
tic % to measure the CPU time of the algorithm
load('all_input_data_flat.mat');

% load the pre-trained node space
load('soinn_trained_assoc.mat');
```

```
% Start with a key/control vector
[CDCnt CDLen] = size(Control_Vec);

AssocClassConnMatrix = zeros (10,10);
RespClass  = zeros (10,10);

for j = 1:CDCnt
    % Here we find which class a given Control Vector belongs to
    j
    [MinClassCnt MinNodeCnt MinDistCnt] =
memlayer_classification_v0(Control_Vec(j,:),Class)
    [MinClassRes MinNodeRes MinDistRes] =
memlayer_classification_v0(Response_Vec(j,:),Class)

    % TBD: Update the node space of the class with the information of
the new
    % node

    % Build Association - Step 19,23,26/A-2
    if AssocClassConnMatrix(MinClassCnt,MinClassRes) <= 0
        AssocClassConnMatrix(MinClassCnt,MinClassRes) = 1;
    else
        AssocClassConnMatrix(MinClassCnt,MinClassRes) =
AssocClassConnMatrix(MinClassCnt,MinClassRes) + 1;
    end

    % associative index of Node i
    AssocIndxNode(MinClassCnt,MinNodeCnt) = MinNodeRes;
    AssocIndxClass(MinClassCnt,MinNodeCnt) = MinClassRes;

    % Response class of Node i
    RespClass(MinClassCnt,MinClassRes) =
RespClass(MinClassCnt,MinClassRes) + 1;
end

toc
```

Supporting Codes:

- readdata: For creating the training and testing vector for creating memory layer.

```
% Generating train and test data from MNIST data set
clear all
%open the file corresponding to digit
k=1;
for j=[1 2 3 4 5 6 7 8 9 0]
    filename =
strcat('Users/Kamela/Documents/MatLabCodes/Codes_ESOINN/MNIST/data',num
2str(j),'.txt');
```

```
    [fid(k) msg] = fopen(filename,'r');
    filename
    l=1;
    %read in the first training example
    % and store it in a 28x28 size matrix t1
    for i=1:2:100
%    for i=2:2:100
        [data28x28,N]=fread(fid(k),[28 28],'uchar');
        data(l,:) = reshape(data28x28,1,28*28);
        dataX = reshape(data28x28,1,28*28);
        l = l+1;
        %imshow(data28x28');
        %pause(0.5)
    end
    fname = strcat('traindata_p',num2str(k),'.mat');
%    fname = strcat('testdata_p',num2str(k),'.mat');
    save (fname,'data');
    k = k+1;
end
```

- prep_key_response_vector_data: For creating temporal sequence for training and inference.

```
% Generating train and test data
% from MNIST data set
clear all
%open the file corresponding to digit
k=1;
for j=[1 2 3 4 5 6 7 8 9 0]
    filename =
strcat('Users/Kamela/Documents/MatLabCodes/Codes_ESOINN/MNIST/data',num
2str(j),'.txt');
    [fid(k) msg] = fopen(filename,'r');
    filename
    l=1;
    %read in the first training example
     % and store it in a 28x28 size matrix t1
    for i=1:2:100
        %    for i=2:2:100
        [data28x28,N]=fread(fid(k),[28 28],'uchar');
        data(k,l,:) = reshape(data28x28,1,28*28);
        dataX = reshape(data28x28,1,28*28);
        l = l+1;
    end
    k = k+1;
end


% Create control and response vectors from the training data
```

```
l = 1;
for j=1:50

    Control_Vec(l,:) =  data(1,j,:);
    Response_Vec(l,:) = data(3,j,:);
    l = l+1;
    Control_Vec(l,:) =  data(2,j,:);
    Response_Vec(l,:) = data(4,j,:);
    l = l+1;
    Control_Vec(l,:) =  data(4,j,:);
    Response_Vec(l,:) = data(1,j,:);
    l = l + 1;
end

fname = strcat('all_input_data_flat','.mat');
save (fname,'data','Control_Vec','Response_Vec');
```

- memlayer_classification_v0: To classify a new incoming node. Used in training temporal sequence.

```
function [MinClass MinNode MinDist] =
memlayer_classification_v0(x,Class)

% x = input vector
% Class = Node Space information
% Class =
%
%           Node: [1xn struct]
%      NodeCount: n
%     ConnMatrix: [pxp double]
%        ConnAge: [pxp double]

tic

[a b] = size(Class);

MinDist = 99999;
MinClass = 0;
MinNode=0;

for m=1:b
    Class(m).NodeCount;
    [Ns1 Ns2] = size(Class(m).Node);
    MostVisNode(m) = 1;
    for n=1:Ns2
        dist = distcalcSOINON(Class(m).Node(n).W, x);

        if dist < MinDist
            MinDist = dist;
            MinClass = m;
            MinNode = n;
```

```
        end
    end
end


toc
```

distcalcSOINON.m

```
function z = distcalcSOINON(w,p)

%DIST Euclidean distance weight function.
%  Algorithm
%  The Euclidean distance D between two vectors X and Y is:
%  D = sqrt(sum((x-y).^2))

[S,R] = size(w);
[R2,Q] = size(p);

if (R ~= Q),  error('Inner matrix dimensions of R and Q do not
match.'),end
if (S ~= R2), error('Inner matrix dimensions of S and R2 do not
match.'),end

Tot = 0;

for i = 1:R
    Tot = Tot + (w(i) - p(i))^2;
end
z = sqrt(Tot)/R;
```

Appendix B - AREA, POWER, DELAY ESTIMATION OF EUCLIDEAN DISTANCE PIPELINE AS A CMOS FPGA IN XILINX VIVADO

Power estimation

The chip used for synthesizing the pipeline is

xc7k70tfbg484-3 (active)

area 23mm*23mm = 529mm^2

product family: kintex-7

tool - Vivado 2015.2

frequency of pipeline – 134Mhz

node – 28nm

Process:

The dynamic power of pipeline and the individual blocks in the pipeline were estimated using the Vivado power report tool after synthesis. The power was estimated at 25 % and 100 % toggle rates. The frequency used for the clock was 134MHz.

The static power of the chip remains constant and varies from one FPGA chip to the other. It remains constant because all the blocks in the FPGA are turned on no matter if it is utilized or not. Static power will not change with CLB used, or not. Nor for DSP used or not. And it will change only very slightly for BRAM used or not.

In the FPGA device, almost everything is powered regardless it is used or not.
Dynamic power varies with what is used, and how fast it is clocked, and the signal's toggle rate.

Procedure:

1) Create a new project in Vivado

2) Select the FPGA device and Device family

3) Run the synthesis, by pressing “Run synthesis” button in the flow navigator

4) Ensure that the RTL schematic is the same the synthesized schematic

5) Ensure that the post synthesis functional simulation works

6) In the Edit timing constraint options, create a clock for the design and assign it to the clock pin

7) Set the period of the clock

8) The period of the clock is chosen in such a way that it is the maximum frequency of the design at which there is no negative slack

9) Run the “Report power” option to generate the power report for the synthesized design

10) Change the toggle rate in the power report options as per the requirement.

- **CMOS FPGA Pipeline dynamic power at 25% toggle rate**

clocks	0.002W
signals	0.002W
logic	0.001W
BRAM	0.001W
I/O	0.015W
Total	0.022W

Figure B- 1: Breakdown of Total Dynamic Power Consumption at 25% Toggle Rate.

power(W)	Name	Clocks (W)	Signals (W)	Data (W)	ck Enable (	Logic (W)	BRAM (W)	I/O (W)
0.0001397993	18-bit mux	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
0.0002292716	18-bit comparator	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
0.0002787289	18-bit adder	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
0.0008572843	8-bit subtractor	<0.0001	<0.0001	<0.0001	<0.0001	0.001	<0.0001	<0.0001
0.0014189907	18-bit accumulator	0.001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
0.0016907840	16-bit LUT Sq. RAM	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	0.001	<0.0001

Figure B- 2: Block level dynamic power consumption at 25% toggle rate.

power(W)	Name	Clocks (W)	Signals (W)	Data (W)	ck Enable (	Logic (W)	BRAM (W)	I/O (W)
0.0001397993	Mux18 (n_nit_2to1_mux)	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
0.0001821431	Ctrl_FSM (FSM)	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
0.0002292716	Comp18 (comparator)	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
0.0002787289	Adder1 (n_bit_adder)	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
0.0003151012	Reg0_siso1 (SISO__1)	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
0.0003151012	Reg0_siso2 (SISO)	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
0.0008572843	Subtracter (n_bit_subtractor)	<0.0001	<0.0001	<0.0001	<0.0001	0.001	<0.0001	<0.0001
0.0013570703	Reg3_18bit_2 (register18bit_init)	0.001	0.001	0.001	<0.0001	<0.0001	<0.0001	<0.0001
0.0014189907	Reg2_accum (accumulator_18bit)	0.001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
0.0016907840	Reg1_SQ2_lut (mdROM_LUT_SQ2)	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	0.001	<0.0001
0.0154921468	IO							

Figure B- 3: Component level dynamic power consumption at 25% toggle rate.

Total Dynamic Power: 0.0222 (W).

- **CMOS FPGA Pipeline dynamic power at 100% toggle rate**

clocks	0.002W
signals	0.002W
logic	0.002W
BRAM	0.001W
I/O	0.016W
Total	0.024W

Figure B- 4: Breakdown of Total Dynamic Power Consumption at 100% Toggle Rate.

power(W)	Name	Clocks (W)	Signals (W)	Data (W)	ck Enable (	Logic (W)	BRAM (W)	I/O (W)
0.0001391754	18-bit mux	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
0.0002279293	18-bit comparator	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
0.0002795308	18-bit adder	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
0.0013834825	8-bit subtractor	<0.0001	<0.0001	<0.0001	<0.0001	0.001	<0.0001	<0.0001
0.0014195796	18-bit accumulator	0.001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
0.0016914924	16-bit LUT Sq. RAM	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	0.001	<0.0001

Figure B- 5: Block level dynamic power consumption at 100% toggle rate.

power(W)	Name	Clocks (W)	Signals (W)	Data (W)	ck Enable (	Logic (W)	BRAM (W)	I/O (W)
0.0001391754	Mux18 (n_nit_2to1_mux)	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
0.0001821431	Ctrl_FSM (FSM)	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
0.0002279293	Comp18 (comparator)	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
0.0002795308	Adder1 (n_bit_adder)	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
0.0005472493	Reg0_siso1 (SISO__1)	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
0.0005472493	Reg0_siso2 (SISO)	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
0.0013542611	Reg3_18bit_2 (register18bit_init)	0.001	0.001	0.001	<0.0001	<0.0001	<0.0001	<0.0001
0.0013834825	Subtracter (n_bit_subtractor)	<0.0001	<0.0001	<0.0001	<0.0001	0.001	<0.0001	<0.0001
0.0014195796	Reg2_accum (accumulator_18bit)	0.001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
0.0016914924	Reg1_SQ2_lut (mdROM_LUT_SQ2)	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	0.001	<0.0001
0.0163488574	IO							

Figure B- 6: Component level dynamic power consumption at 100% toggle rate.

Total Dynamic Power: 0.0241 (W).

Delay Estimation

Process:

The delay values were estimated by using Timing report tool after synthesis of the design. The frequency used for the clock is 134MHz.

Procedure:

1) Create a new project in Vivado

2) Select the FPGA device and Device family

3) Run the synthesis, by pressing "Run synthesis" button in the flow navigator

4) Ensure that the RTL schematic is the same as the synthesized schematic

5) Ensure that the post synthesis functional simulation works

6) In the Edit timing constraint options, create a clock for the design and assign it to the clock pin

7) Set the period of the clock

8) The period of the clock is chosen in such a way that it is the maximum frequency (134MHz) of the design at which there is no negative slack

9) Run the "Report Timing summary" option to generate the timing summary of the design and various logic blocks used.

Name	logic delay (ns)	net delay (ns)	total delay (ns)	freq(Mhz)
18-bit mux	2.98	0.867	3.847	259.9428
18-bit comparator	3.56	1.06	4.62	216.4502
18-bit adder	2.29	0.76	3.05	327.8689
8-bit subtractor	1.25	5.7	6.95	143.8849
18-bit accumulator	2.51	0.434	2.944	339.6739
16-bit LUT Sq. RAM	3.78	0.71	4.49	222.7171

Figure B- 7: Delay breakdown of various components in CMOS FPGA.

Area estimation

The Vivado tool reports the Utilization (after synthesis) of resources for the entire pipeline as shown below. It is done using the Vivado utilization report after synthesis-

Resource	Estimation	Available	Utilization %
FF	76	82000	0.37
LUT	100	41000	0.96
I/O	36	285	33.96
BRAM	0.50	135	2.00
BUFG	1	32	3.13

Figure B- 8: Area Utilization Breakdown.

Components that consume the chip area are-

IO count = 484

IOBs = 285

LUT = 41000

FF = 82000

BRAM = 135

DSP = 240

Transceivers = 8

PCIE = 1

MMCMs = 6

Assuming that all the available units occupy 100% of the chip, we can estimate the area of the pipeline-

(Total utilization/ Total available units) * 100

From the table total utilization = 76+100+36+0.5+1 = 213.5

Total available units = 484+285+41000+82000+135+240+8+1+6= 124159

Let's say that 124159 units consume 529mm^2 of the chip.

Percentage area occupied by the pipeline is = (213.5/124159) * 100 = 0.171%

Approximate estimation of the Area in mm^2 is = 0.171% of 529mm^2 = 0.00171 * 529 = 0.904mm^2.

Name	Slice LUTs	Slice Registers	Block RAM Tile	Bonded IOB	BUFGCTRL
min_dist_pipeline	100	76	0.5	36	1
Adder1 (n_bit_adder)	18	0	0	0	0
Comp18 (comparator)	10	0	0	0	0
Ctrl_FSM (FSM)	2	5	0	0	0
Mux18 (n_nit_2to1_mux)	9	0	0	0	0
Reg0_siso1 (SISO__1)	0	8	0	0	0
Reg0_siso2 (SISO)	0	8	0	0	0
Reg1_SQ2_lut (mdROM_LUT_SQ2)	0	0	0.5	0	0
Reg2_accum (accumulator_18bit)	12	37	0	0	0
Reg3_18bit_2 (register18bit_init)	1	18	0	0	0
Subtracter (n_bit_subtractor)	48	0	0	0	0

Figure B- 9: Area Utilization by Various Components.

From the above table:

LUT occupy (100/213.5 *100) =46.8% of 0.904mm^2 = 0.423mm^2

1) Adder uses 18 LUTs = 18/100 *100 = 18% of LUT area = 0.18*0.423= 0.076mm^2

2) subtractor uses 48 LUT = 48/100 *100 = 48% of LUT area = 0.48*0.423= 0.203mm^2

3) comparator uses 10 LUT = 10/100 *100 = 10% of LUT area = 0.1 * 0.423= 0.0423mm^2

4) LUT SQ RAM uses 0.5 BRAM.

BRAM occupy 0.5/213.5 *100 = 0.23% of total area = 0.0023 * 0.904 = 0.00207mm^2

LUT SQ RAM area = 0.00207mm^2

5) MUX uses 9 LUT = 9/100 *100 = 9% of LUT area = 0.09*0.423= 0.038mm^2

6) Accumulator uses 12 LUT and 37 registers

Area occupied by LUT is 12/100*100= 12% of LUT area = 0.12*0.423= 0.05mm^2

There are 76 registers and it occupy (76/213.5 *100) =35.5% of total area = 0.355*0.904 = 0.32mm^2

Accumulator uses 37 registers = 37/76*100 = 48.6% of register area.

Accumulator Register area= 0.486 * 0.32 = 0.15mm^2

Total area occupied by registers = 0.05mm^2 + 0.15mm^2= 0.2mm^2

Adding area of individual blocks would approximately be equal to the area of the pipeline $\sim$ 0.6 mm^2.

Appendix C - SPACE-TIME NOTATION BASED CIRCUIT DRAWING

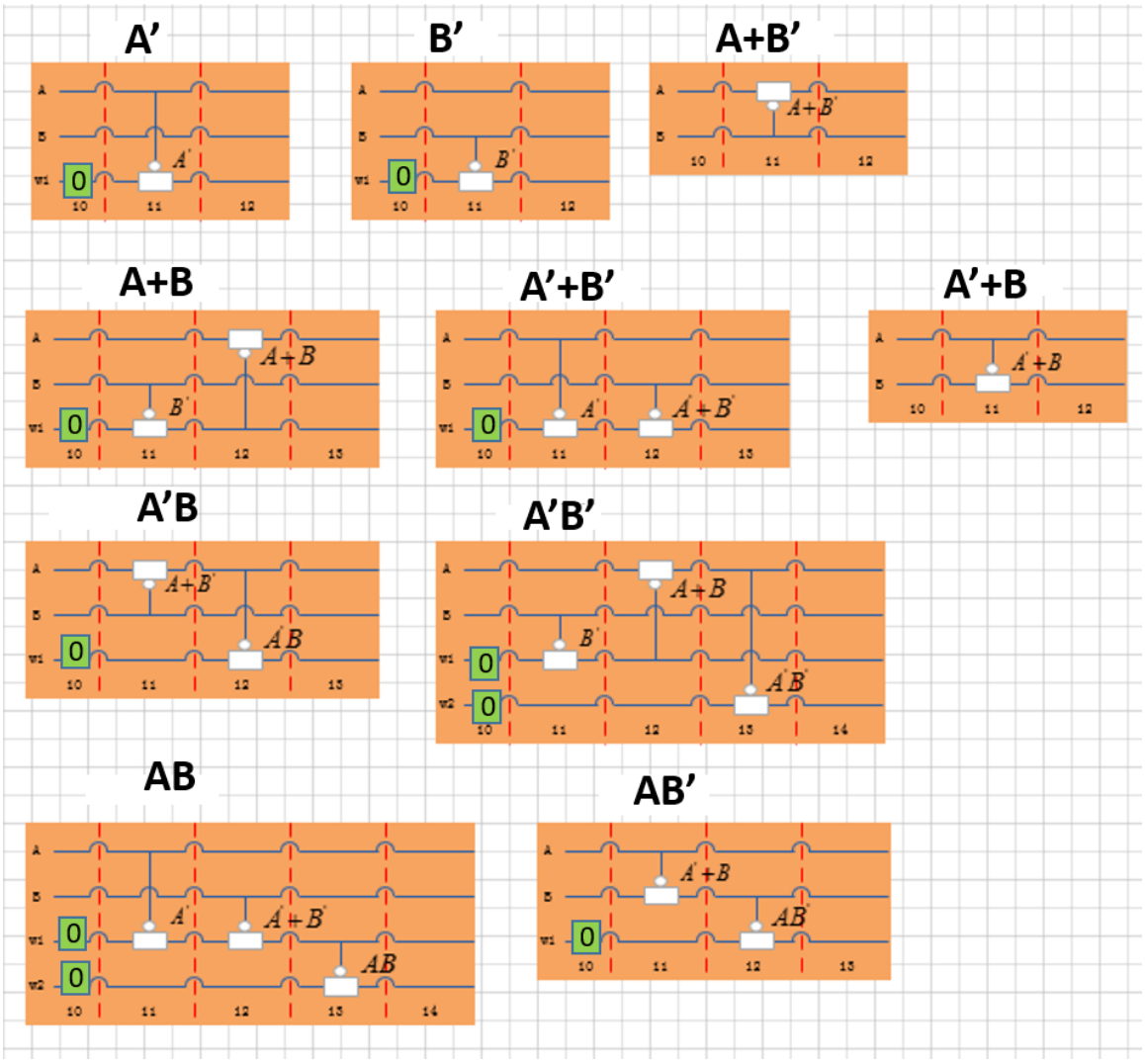


Figure C- 1 Basic Logic Synthesis.

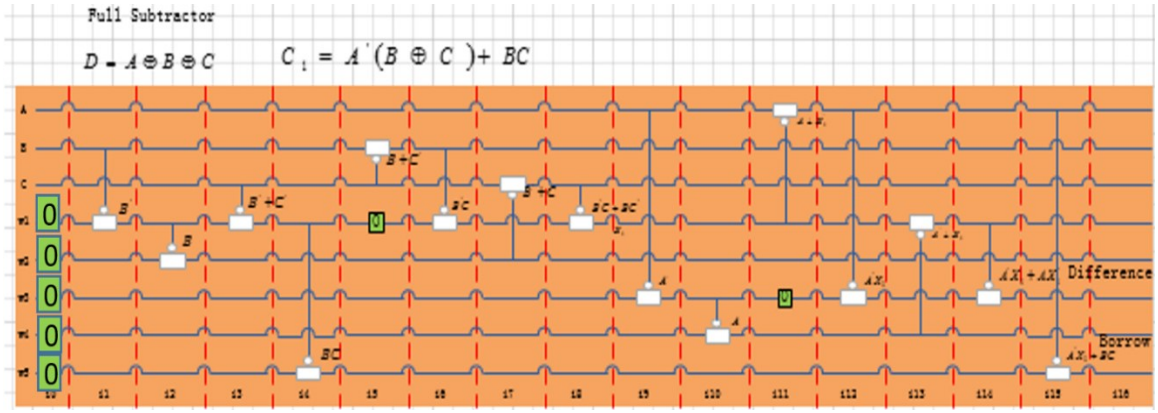


Figure C- 2: 1-bit Full Subtractor Design.

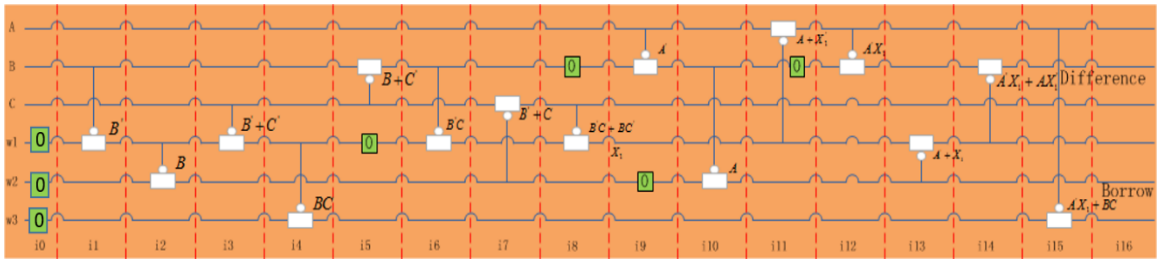


Figure C- 3: 1-bit Full Subtractor Design (Optimized for area).

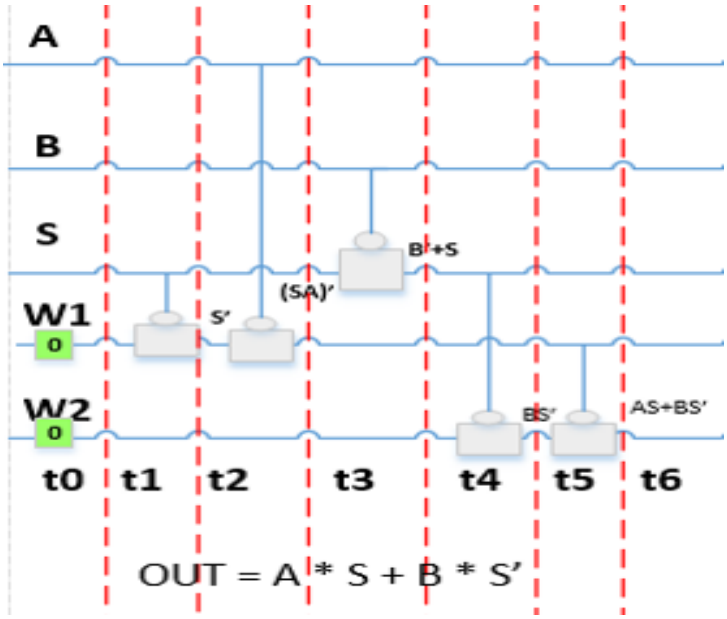


Figure C- 4: 2-to-1 Multiplexer Design.

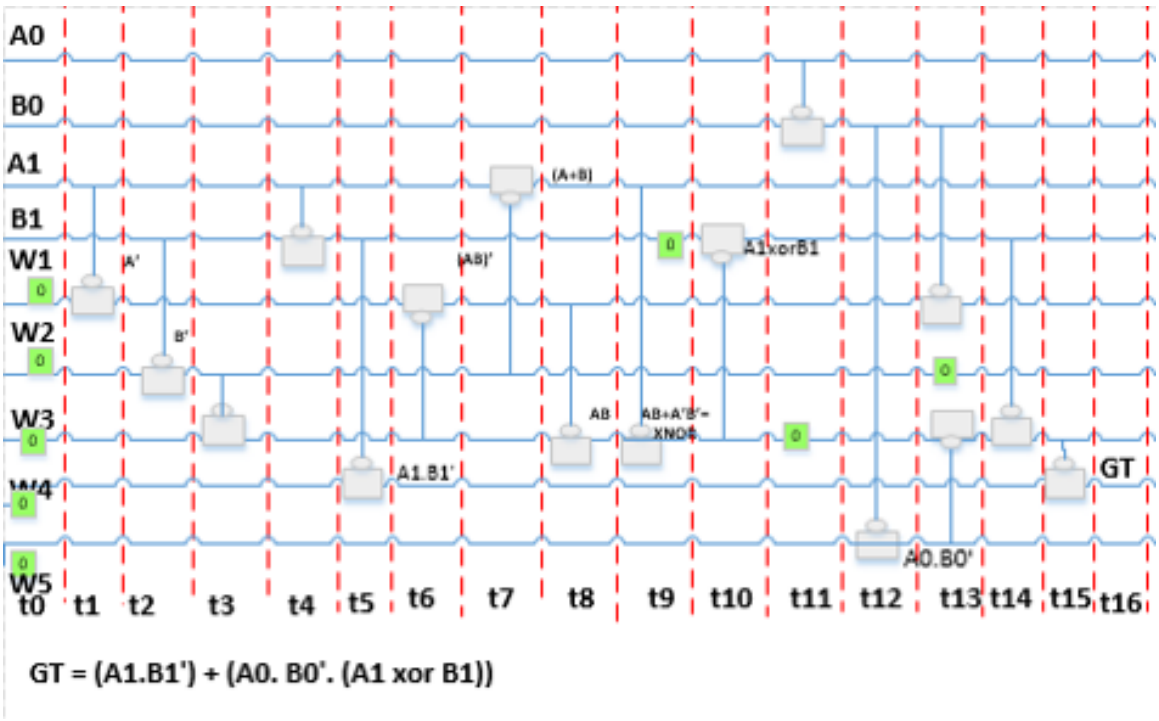


Figure C- 5: 5: 2-bit Greater than (GT) Comparator Design.

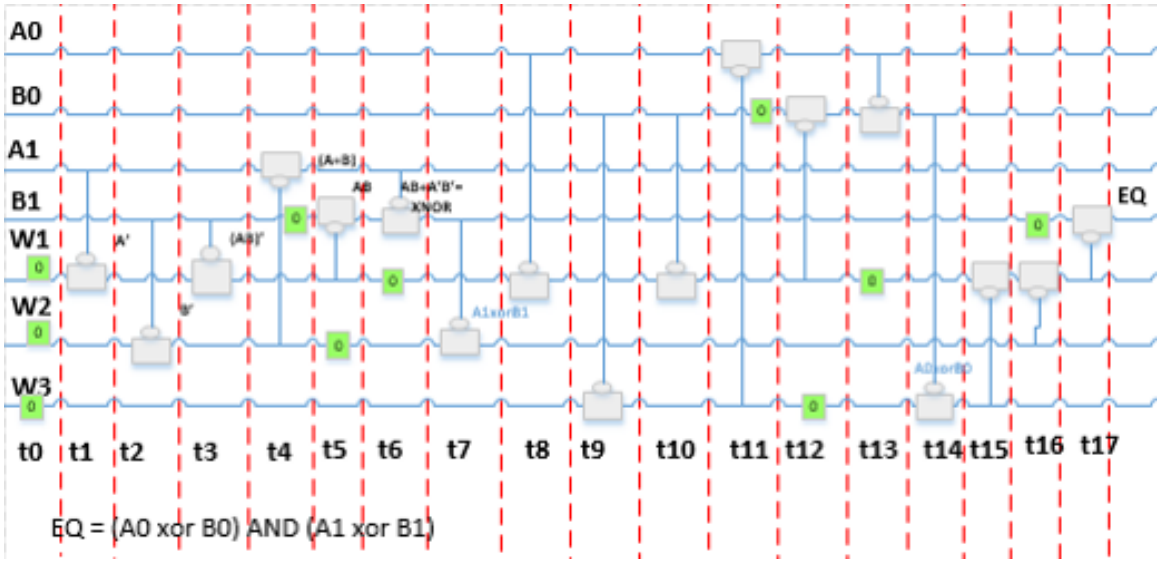


Figure C- 6: 2-bit Equal Comparator Design.